IDENTIFICATION DES OBJETS DANS UN CODE PROCÉDURAL BASÉE SUR LA DÉCOMPOSITION DE GRAPHES

par

François Dumont

mémoire présenté au Département de mathématiques et d'informatique

en vue de l'obtention du grade de maître ès sciences (M.Sc.)

FACULTÉ DES SCIENCES

UNIVERSITÉ DE SHERBROOKE

Sherbrooke, Québec, Canada, décembre 1997

Le 18/8/98 , le jury suivant a accepté ce mémoire dans sa version finale.
date

Président-rapporteur: M. Froduald Kabanza
Département de mathématiques et d'informatique

Membre: M. Richard St-Denis
Département de mathématiques et d'informatique

Membre: M. Marc Frappier
Département de mathématiques et d'informatique

SOMMAIRE

Les systèmes informatiques à forte entropie, développés avec une approche procédurale, ont subi beaucoup de modifications au cours des années. Par conséquent, ils sont devenus complexes et très mal documentés. De plus, la maintenance de ces systèmes est difficile à assurer et très coûteuse. Afin de pallier ces difficultés, plusieurs organisations orientent leurs systèmes vers de nouvelles technologies. Dans ces systèmes à forte entropie, l'identification des objets est essentielle pour conduire ceux-ci vers la technologie orientée objets. Cette technologie favorise la réutilisation, l'extension, la flexibilité, l'encapsulation, la modularité et la maintenance.

Dans ce mémoire, nous présentons une approche automatique permettant d'identifier les objets dans un code procédural. L'identification des objets est la première phase de la migration d'un code source procédural vers un code source orienté objets. L'approche suggérée est basée sur la décomposition de graphes bipartites. Notre approche consiste à identifier des sous-graphes connexes à l'intérieur du graphe d'un système. Chacun des sous-graphes connexes est composé d'un noeud représentant les données, et d'un ou plusieurs noeuds représentant les méthodes de l'objet. Les sous-graphes connexes représentent des candidats objets du système procédural. Cette approche est une amélioration de celle présentée par Canfora, Cimitile et Munro.

Nous avons appliqué notre approche d'identification des objets sur des systèmes de grandes et moyennes tailles. Les résultats démontrent que l'approche est capable d'identifier des objets. De plus, un exemple connu illustre l'approche. Finalement, nous suggérons des pistes pour des futurs travaux.

REMERCIEMENTS

La réalisation de ces travaux de recherche fut une expérience très enrichissante. Cependant, sans l'appui et l'assistance de plusieurs personnes, j'aurais eu beaucoup plus de difficultés à réaliser ce travail.

Mon directeur de recherche, M. Marc Frappier, fut un modèle de qui j'ai énormément appris. De fait, sa grande patience, son professionnalisme, son côté méthodique et scientifique m'ont permis de développer les qualités requises par la réalisation de ce mémoire. Je désire remercier le Centre de recherche informatique de Montréal pour le support financier qu'il m'a accordé afin que je réalise ces travaux de recherche. En outre, je tiens à souligner l'excellent travail de mon superviseur au CRIM, M. Houari Sahraoui. De plus, je tiens à remercier sincèrement les gens de mon équipe de travail pour leur aide et leur support lors de mon séjour au CRIM, soit Emna, Hakim, Judith, Michel et Nathalie. Aussi, j'aimerais remercier M. Serge Oligny, qui a grandement influencé ma décision d'entreprendre des études de deuxième cycle. Le support et l'encouragement de mes amis, Pascal, Janick, Pierre, Robin et beaucoup d'autres, ont rendu cette expérience plus agréable. Le personnel du Département de mathématiques et d'informatique fut toujours d'une grande disponibilité et amabilité. C'est avec une grande sincérité que je remercie tous ces gens. Enfin, mon père fut un exemple de support et d'encouragement à plusieurs points de vue. Sa présence à mes côtés a joué un rôle important dans la réalisation de ce travail. Il mérite toute ma gratitude et mon estime.

TABLE DES MATIÈRES

SOMMAIRE ii

REMERCIEMENTS iii

TABLE DES MATIÈRES iv

LISTE DES TABLEAUX viii

LISTE DES FIGURES ix

INTRODUCTION 1

La problématique et l'approche 1

Les raisons de résoudre ce problème 4

Nos contributions 7

L'organisation du mémoire 8

1 L'ÉTAT DE L'ART 10

1.1 Les définitions 10

1.2 Le paradigme objet 13

1.3 La rétro-ingénierie des logiciels 15

1.3.1 Les approches classiques 16

1.3.2 Les approches utilisant les réseaux neuronaux 20

1.4 L'identification des objets 23

1.4.1 La compréhension des programmes 23
1.4.2 L'abstraction des programmes 24
1.4.3 La démarche d'identification des objets 26

2 LA SOLUTION **35**

2.1 L'algorithme d'identification des objets 35
2.1.1 Le graphe de références 36
2.1.2 Les éléments nécessaires à l'application de l'algorithme 38
2.1.3 Le seuil 40
2.1.4 L'algorithme 40
2.2 La focalisation 42
2.2.1 La focalisation de base 46
2.3 Les problèmes identifiés avec l'approche 46
2.3.1 L'identification des classes d'objets 46
2.3.2 L'utilisation des variables globales 47
2.3.3 Le calcul du seuil 47
2.3.4 La focalisation systématique 47
2.4 Nos contributions 48
2.4.1 Le graphe de visibilité des types 48
2.4.2 Le graphe de visibilité des données 53
2.4.3 Le graphe de références amélioré 59
2.4.4 Notre manière de calculer le seuil 61
2.4.5 Une modification de la condition de fin de l'algorithme 61

3 LA RÉALISATION **62**

3.1 Les extracteurs de graphes 62
3.1.1 L'analyse du code source 63
3.1.2 Les règles générales aux trois extracteurs de graphes 63
3.1.3 L'extracteur du graphe de références 64

3.1.4 L'extracteur du graphe de références amélioré 65
3.1.5 L'extracteur du graphe de visibilité des types 65
3.1.6 L'extracteur du graphe de visibilité des données 67
3.2 L'algorithme d'identification des objets dans un code source procédural . 69
3.2.1 La procédure principale 69
3.2.2 Extraire l'information de la base de faits 70
3.2.3 Créer le graphe à partir des informations extraites 71
3.2.4 Afficher le graphe 72
3.2.5 Calculer le $\Delta IC(F)$ 72
3.2.6 Calculer le seuil 73
3.2.7 Calculer l'ensemble REGROUPER 74
3.2.8 Calculer l'ensemble FOCALISER 74
3.2.9 Afficher les ensembles REGROUPER et FOCALISER 74
3.2.10 Effectuer le regroupement 74
3.2.11 Effectuer la focalisation 75
3.3 L'exemple 77
3.3.1 La base de faits 77
3.3.2 L'algorithme d'identification des objets 79

4 L'ÉTUDE DE CAS **86**
4.1 Le système SBC1 86
4.2 Le système SBC2 87
4.3 Le système SBC3 88
4.4 La validation des résultats 88
4.4.1 La validation de SBC1 89
4.4.2 La validation de SBC2 90
4.4.3 La validation de SBC3 92

CONCLUSION **93**

Les forces de la solution 94

Les faiblesses de la solution 94

Le prolongement des travaux 95

ANNEXES **96**

Annexe A - Exemple1.cc **96**

Annexe B - Exemple2.cc **102**

Annexe C - Exemple3.cc **107**

BIBLIOGRAPHIE **115**

LISTE DES TABLEAUX

1.1 Exemple de la classe animal. 14

1.2 Objectifs de la rétro-ingénierie des logiciels. 16

1.3 Phases du paradigme RE^2. 17

1.4 Approche de Gall et al. 31

1.5 Approche de Shin. 32

1.6 Approche de Canfora et al. 33

1.7 Approche de Yeh et al. 34

4.1 Objets identifiés dans le système SBC1. 87

4.2 Objets identifiés dans le système SBC2. 87

4.3 Objets identifiés dans le système SBC3. 88

4.4 Validation des résultats de SBC1. 90

4.5 Validation des résultats de SBC2. 91

4.6 Validation des résultats de SBC3. 92

LISTE DES FIGURES

1 Exemple d'un graphe de références. 3

2 Exemple d'un graphe de références ayant des mauvaises connexions. 4

1.1 La rétro-ingénierie dans le cycle de développement d'un logiciel. 11

1.2 L'ingénierie dans le cycle de développement d'un logiciel. 11

1.3 La réingénierie dans le cycle de développement d'un logiciel. 12

1.4 Graphe de références du programme `Exemple1.cc`. 25

1.5 Graphe de visibilité des types du programme `Exemple1.cc`. 26

1.6 Graphe d'appels des fonctions du programme `Exemple1.cc`. 27

2.1 Exemple d'un graphe *Def/Use* intraprocédural. 45

2.2 Le graphe de visibilité des types après l'application de l'algorithme. 51

2.3 Le graphe de visibilité des types avec l'intervention d'un expert lors de l'application de l'algorithme. 54

2.4 Le graphe de visibilité des données du programme `Exemple2.cc`. 56

2.5 Le graphe de visibilité des données après l'application de l'algorithme. 57

2.6 Les regroupements possibles sur le graphe de visibilité des données. 58

2.7 Les candidats objets obtenus avec l'intervention d'un expert lors de l'application de l'algorithme. 59

2.8 Exemple d'une focalisation non systématique. 60

3.1 Graphe de références du programme `Exemple3.cc`. 80

3.2 Graphe de références après la première itération. 83

3.3 Graphe de références après la deuxième itération. 85
3.4 Les objets identifiés dans le programme `Exemple3.cc`. 85

INTRODUCTION

Le génie logiciel est une discipline dont l'objectif est de produire, de manière efficace, des logiciels de haute qualité. En 1968, à la conférence de Garmish, une réflexion sur la qualité du logiciel et la productivité des ingénieurs a mené à la naissance du terme génie logiciel. Selon Barazin [26], le génie logiciel est l'application de la science et des mathématiques au processus par lequel on rend les ordinateurs utiles aux personnes. Par ailleurs, Bauer [26] présente le génie logiciel comme étant l'établissement et l'utilisation de principes de génie solides pour produire économiquement des logiciels qui sont fiables et efficaces. En considérant les deux définitions précédentes, on peut conclure que l'objectif du génie logiciel est de définir des outils et des méthodes scientifiques pour produire des logiciels répondant aux besoins des utilisateurs en respectant les contraintes de temps et de budget.

Ce mémoire s'intéresse à une sous-discipline du génie logiciel. Cette sous-discipline est la réingénierie de programmes procéduraux vers une technologie orientée objets.

La problématique et l'approche

Le problème résolu dans ce mémoire est l'identification des objets dans un code source procédural, ce qui peut permettre de migrer une application écrite dans un langage procédural vers une application écrite dans un langage orienté objets. Pour résoudre ce

problème, nous avons identifié deux approches : l'approche dite dépendante du domaine d'application et l'approche dite indépendante du domaine d'application.

L'approche dite dépendante du domaine d'application demande des connaissances d'un expert pour définir un modèle d'application. Ce modèle, utilisé conjointement avec le code source procédural de l'application, permet de recouvrer la conception pour ensuite être capable d'identifier les objets de cette application.

L'approche dite indépendante du domaine d'application ne demande aucune expertise du domaine de l'application et utilise uniquement le code source de l'application pour identifier les objets. Par conséquent, le niveau d'automatisation du processus permettant l'identification des objets est beaucoup plus élevé avec l'approche indépendante du domaine qu'avec l'approche dépendante du domaine.

Nous avons choisi l'approche indépendante du domaine pour solutionner le problème, car elle peut être automatisée. Cette automatisation permet d'obtenir un processus d'identification des objets qui est économique et efficace.

L'algorithme que nous proposons est inspiré des travaux de Canfora et al. [4]. Cet algorithme utilise un graphe orienté bipartite appelé "graphe de références". Le premier ensemble de noeuds représente les variables globales contenues dans le code source de l'application. Le deuxième ensemble de noeuds représente les fonctions de cette même application. Les arcs permettent de lier les noeuds de type variables globales aux noeuds de type fonctions. Ces arcs indiquent qu'une fonction fait référence à une variable globale. La figure 1 montre un exemple d'un graphe de références.

L'idée de base de l'algorithme est que les sous-graphes connexes à l'intérieur du graphe de références représentent de bons candidats pour créer des objets. Il y a cependant des sous-graphes connexes qui ne sont pas de bons candidats pour créer des objets. En effet, ces sous-graphes regroupent des fonctions qui n'ont pas besoin d'appartenir au même

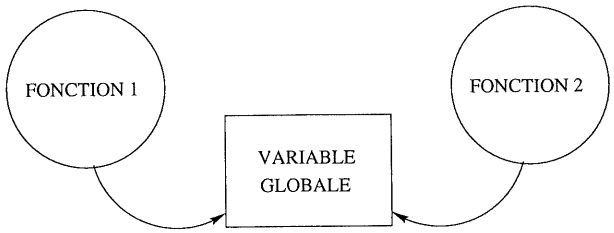

FIG. 1: Exemple d'un graphe de références.

objet pour avoir une cohésion fonctionnelle. Il est possible d'éliminer ces connexions indésirables en brisant le sous-graphe en plusieurs sous-graphes. Les nouveaux sous-graphes connexes ont une meilleure cohésion fonctionnelle. Les connexions indésirables peuvent être causées principalement par deux situations.

Le premier type de connexions indésirables concerne les fonctions ayant une faible cohésion. Ces fonctions manipulent des éléments de données qui n'ont pas de liens logiques entre eux. Par conséquent, ces éléments de données ne doivent pas se retrouver dans le même objet. Dans l'exemple illustré à la figure 2, les connexions indésirables représentant cette situation sont les connexions 1, 2 et 3. Les variables globales `PILE`, `FILE` et `CLIENT` n'ont pas de liens entre elles. Par contre, la fonction `INITIALISER` fait référence à ces trois variables. Donc, il est possible de conclure que la fonction `INITIALISER` a une faible cohésion. Il faut éliminer ce type de connexion lors de la détermination des candidats appelés à devenir des objets.

La seconde situation est causée habituellement par une mauvaise programmation. Il s'agit de variables qui sont déclarées globales mais qui devraient plûtot être déclarées locales à chaque fonction. Évidemment, ces variables globales ne devraient pas faire partie d'un objet. Dans la figure 2, le programmeur a déclaré la variable `INDICE` globale, puisqu'elle est utilisée dans les fonctions `EMPILER`, `ENFILER` et `AJOUTER CLIENT`. Donc, il faut éliminer les connexions 7, 8 et 9 lors de la détermination des candidats à être élus comme objets.

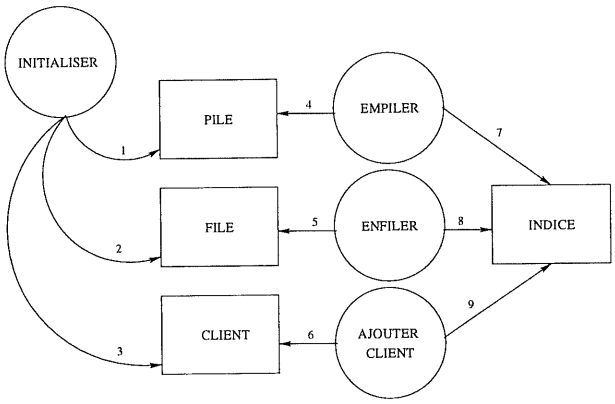

FIG. 2: Exemple d'un graphe de références ayant des mauvaises connexions.

Dans l'exemple précédent, il est facile de voir les connexions indésirables. Par contre, dans des programmes de plusieurs milliers de lignes ayant des noms de variables et de fonctions non significatifs, ce n'est pas aussi simple. Dans ce mémoire, nous proposons une technique pour éliminer ces connexions indésirables et obtenir des sous-graphes connexes significatifs pour un humain.

Les raisons de résoudre ce problème

Cette section présente les principales raisons qui justifient la migration vers une architecture orientée objets. Il y a quatre secteurs où cette migration vers l'orienté objets s'avère très intéressante : la maintenance, l'évolution des logiciels, l'évolution du matériel et la réutilisation.

La maintenance

Une étude réalisée vers la fin des années 1980 par Osborne [11] du *National Bureau of Standards* suggère que 60 % à 85 % du coût total d'un logiciel est dû à la maintenance. Par ailleurs, en 1989, Corbi [21] affirme qu'une très grande partie de l'argent investi pour

assurer la maintenance des systèmes est dépensé uniquement pour essayer de comprendre ces systèmes. En effet, la compréhension des systèmes informatiques représente environ 50 % de l'effort requis pour assurer la maintenance. Il existe quatre types de maintenance [2, 15, 24] :

corrective : la correction des erreurs signalées par un utilisateur;

adaptative : l'adaptation du logiciel à un changement d'environnement;

perfective : l'ajout de nouvelles fonctionnalités au logiciel;

préventive : la prévention des erreurs par l'anticipation des changements à venir.

Les programmes orientés objets sont plus faciles à maintenir que les programmes procéduraux, car ils sont généralement plus simples à comprendre. La cohésion d'un logiciel correspond à la capacité de chaque fonction de réaliser une seule tâche. Un logiciel ayant une forte cohésion peut être plus simple à comprendre qu'un logiciel ayant une faible cohésion. Par ailleurs, le couplage est la mesure du degré d'interconnexions entre les modules d'un logiciel. Un logiciel ayant un faible couplage indique qu'il y a peu d'échanges de données entre ses modules. Donc, un logiciel ayant un couplage faible peut être plus simple à comprendre qu'un logiciel ayant un couplage élevé. L'indépendance fonctionnelle est atteinte en développant des modules ayant un faible couplage et une forte cohésion. Par ailleurs, l'orienté objets favorise la création de programmes modulaires. La modularité est définie comme étant l'indépendance fonctionnelle entre les composantes d'un programme [17]. Des modules indépendants sont plus faciles à maintenir, car il y a beaucoup moins d'effets secondaires causés par la modification du code source. Selon Myers [17], la modularité est le seul attribut d'un logiciel qui lui permet d'être intellectuellement gérable. En effet, un logiciel monolithique (un seul gros module) est très difficile à comprendre, car au lieu d'avoir plusieurs petits problèmes à résoudre, il n'en a qu'un seul, mais il est énorme.

Par ailleurs, l'encapsulation est une caractéristique propre aux programmes orientés objets. L'encapsulation se définit comme un principe permettant de diminuer la

complexité des logiciels. L'encapsulation permet de cacher certains détails de manière à diminuer l'effort requis pour la compréhension des logiciels. Les objets permettent d'encapsuler de l'information inutile aux autres objets.

L'évolution des logiciels

Les systèmes à forte entropie [1] sont souvent la seule source de documentation sur les règles de gestion des entreprises. Les systèmes à forte entropie sont des systèmes habituellement âgés dont la structure s'est dégradée à la suite de nombreuses modifications [21]. De plus, il s'agit de systèmes procéduraux développés avant la venue de la programmation structurée et des principes du génie logiciel. Dans plusieurs cas, les gens qui ont implanté ces règles de gestion ne sont plus disponibles pour assurer leur maintenance.

Les systèmes à forte entropie peuvent difficilement évoluer. Il existe tout de même deux façons de faciliter leur évolution. La première solution consiste à récrire manuellement ces systèmes dans un langage de programmation plus évolué en suivant les principes du génie logiciel. Cette réécriture est très coûteuse à réaliser, car il faut franchir toutes les étapes du cycle de développement d'un logiciel. La seconde solution consiste à utiliser des outils de GLAO (Génie Logiciel Assisté par Ordinateur) pour traduire les systèmes à forte entropie en systèmes écrits dans des langages plus évolués. Cette réingénierie est beaucoup moins coûteuse que la réécriture manuelle du logiciel, car l'utilisation d'un outil pour extraire les règles de gestion du code source peut permettre d'économiser beaucoup de temps.

L'évolution du matériel

Les systèmes à forte entropie sont utilisés dans les ordinateurs des générations précédentes. Pour demeurer compétitives, les entreprises doivent suivre l'évolution technolo-

[1] Traduction libre de *Legacy Systems*.

gique. Il va de soi que cette évolution matérielle doit être suivie d'une évolution des logiciels. Dans ce cas, il faut convertir les systèmes pour les adapter aux nouveaux environnements. Conformément à ce qui précède, cette transformation des logiciels peut être réalisée de deux manières différentes : soit par la réécriture manuelle, soit par la réingénierie de ces systèmes dans un langage de programmation plus évolué.

La réutilisation

La réutilisation du code source de systèmes à forte entropie est très limitée. En effet, il est très difficile de comprendre le fonctionnement de ces systèmes informatiques, car leur code source est très volumineux et très peu structuré. Selon Presseman [17], la réutilisabilité est une caractéristique importante d'une composante logicielle de haute qualité. Il est plus facile de réutiliser du code source si les données et les traitements sont encapsulés dans un objet.

Nos contributions

Nous avons apporté des améliorations à l'approche présentée par Canfora et al. [4]. Premièrement, cette approche ne permet pas d'identifier des classes d'objets. Pour tenter de solutionner ce problème, nous proposons d'utiliser un autre type de graphe comme entrée à l'algorithme d'identification des objets, c'est-à-dire le graphe de visibilité des types.

Deuxièmement, l'approche présentée par Canfora et al. [4] est basée sur les variables globales. Donc, il est impossible d'identifier les objets d'une librairie de fonctions ou d'un programme ayant peu de variables globales. Pour pallier ce problème, nous proposons d'utiliser un nouveau type de graphe comme entrée à l'algorithme d'identification des objets, c'est-à-dire le graphe de visibilité des données.

Troisièmement, l'approche présentée par Canfora et al. [4] focalise systématiquement certaines fonctions pour parvenir à identifier les objets d'un programme procédural. En réalité, la focalisation d'une fonction n'est pas toujours possible à réaliser. Par conséquent, nous proposons d'utiliser un nouveau type de graphe comme entrée à l'algorithme d'identification des objets, c'est-à-dire le graphe de références amélioré. C'est un graphe de références permettant de tenir compte du mode d'utilisation d'une variable globale par une fonction lors de la focalisation.

Quatrièmement, l'approche présentée par Canfora et al. [4] demande de calculer un seuil permettant d'effectuer des modifications sur le graphe d'un programme procédural. Ces modifications permettent d'obtenir des sous-graphes connexes représentant des candidats objets. Leur façon de calculer le seuil est difficilement automatisable, car elle est basée sur le style de programmation. Nous proposons une façon automatique de calculer le seuil permettant d'obtenir des objets de petite taille.

Cinquièmement, nous avons modifié la condition de fin de l'algorithme présenté par Canfora et al. [4]. L'identification des objets dans un système réel a permis de découvrir que l'algorithme ne terminait pas pour certains cas particuliers.

Sixièmement, nous avons validé notre approche sur des systèmes réels de grandes et moyennes tailles. Ce type de validation n'a pas été réalisé par Canfora et al. [4].

L'organisation du mémoire

Le chapitre 1 expose l'état de l'art en réingénierie des logiciels. Le chapitre 2 explique la solution qui est proposée pour résoudre le problème. Le chapitre 3 présente le prototype de l'outil permettant l'identification des objets dans un code source procédural. Le chapitre 4 montre les résultats obtenus lors de l'identification des objets de trois systèmes informatiques écrits dans des langages procéduraux. La conclusion nous permet de faire un résumé des contributions et de discuter des problèmes à résoudre à la suite de la

réalisation de cette recherche. Les annexes contiennent respectivement le code source des programmes `Exemple1.cc`, `Exemple2.cc` et `Exemple3.cc`.

Chapitre 1

L'ÉTAT DE L'ART

Ce chapitre porte sur les travaux qui ont été réalisés dans le domaine de la réingénierie des logiciels. Premièrement, nous faisons un survol des principaux termes utilisés en réingénierie des logiciels. Deuxièmement, nous dressons un portrait du paradigme objet pour situer le lecteur. Troisièmement, nous présentons des approches de rétro-ingénierie des logiciels. Quatrièmement, nous étudions les travaux qui ont été réalisés sur l'identification des objets dans un code source procédural.

1.1 Les définitions

Cette section a comme objectif de définir les principaux termes utilisés dans le domaine de la réingénierie des logiciels.

La rétro-ingénierie est le processus par lequel on analyse un système dans le but d'identifier ses composantes et leurs relations. De plus, ce processus crée une représentation du système sous une autre forme ou à un niveau d'abstraction plus élevé. Contrairement à la réingénierie, la rétro-ingénierie est uniquement un processus d'analyse et non un processus de modification [13]. La figure 1.1 présente les processus de rétro-ingénierie des logiciels.

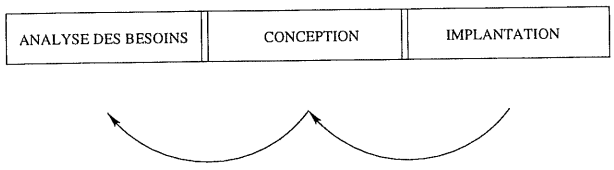

FIG. 1.1: La rétro-ingénierie dans le cycle de développement d'un logiciel.

L'ingénierie est le processus traditionnel permettant de passer d'un haut niveau d'abstraction vers un niveau plus bas. Par exemple, il s'agit de passer du niveau logique d'implantation au niveau physique. En résumé, ce processus comporte trois phases : la phase d'analyse des besoins, la phase de conception et la phase d'implantation du logiciel [13]. La figure 1.2 présente le processus d'ingénierie des logiciels.

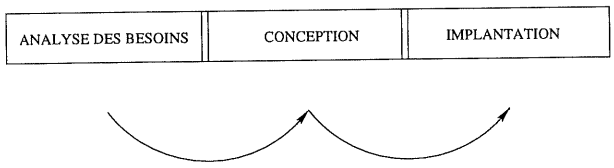

FIG. 1.2: L'ingénierie dans le cycle de développement d'un logiciel.

La restructuration est le processus de transformation d'une représentation d'un système en une autre forme. Habituellement, l'ancienne représentation et la nouvelle ont le même niveau d'abstraction. Cette transformation doit préserver le comportement externe du système, c'est-à-dire que le système doit conserver la même fonctionnalité. Pour effectuer le travail de restructuration, les analystes utilisent certaines approches de la rétro-ingénierie des logiciels. Le terme "restructuration" est devenu populaire avec la transformation de code non-structuré (code spaghetti utilisant des GOTO) en un code structuré. La restructuration est souvent utilisée comme une forme de maintenance préventive permettant d'améliorer la qualité du système en respectant certains standards. Finalement, la restructuration du code peut permettre d'effectuer la maintenance adaptative du système. Comme nous l'avons signalé plus haut, cette forme de maintenance permet au système de ren-

contrer de nouvelles contraintes environnementales. Cette forme de maintenance n'implique aucune réévaluation au niveau des besoins fonctionnels du système [13].

La réingénierie est le processus d'analyse ou de modification d'un système dans le but de le reconstruire sous une nouvelle forme afin d'améliorer sa qualité et sa maintenabilité. Évidemment, la phase d'analyse de ce processus correspond au processus de rétro-ingénierie. De plus, la réingénierie est un processus qui fait appel à des notions de l'ingénierie des logiciels et de la restructuration du code source. Finalement, il est important de remarquer que la réingénierie permet la modification du système pour qu'il comble de nouveaux besoins, des besoins que le système original ne comblait pas [13]. La figure 1.3 présente le processus de réingénierie des logiciels.

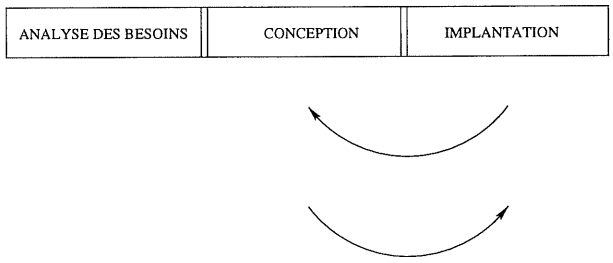

FIG. 1.3: La réingénierie dans le cycle de développement d'un logiciel.

Le recouvrement de la conception est un sous-ensemble de la rétro-ingénierie faisant appel à la connaissance du domaine dans lequel le système est en opération. De plus, il fait appel à toutes les informations externes qui sont disponibles sur le système et à certains raisonnements flous ou déductions concernant celui-ci. Évidemment, ces informations propres au recouvrement de la conception doivent être ajoutées à celles recueillies par la rétro-ingénierie pour identifier un haut niveau d'abstraction qui soit significatif. En effet, ce haut niveau d'abstraction sera meilleur que celui obtenu directement par une simple analyse du code source du système. En

1989, Biggerstaff indique que le but ultime du recouvrement de la conception est la reconstruction d'une conception abstraite à partir d'une combinaison du code source, d'une documentation sur la conception (si elle est disponible), d'expertises personnelles et de connaissances générales sur un problème ou sur un domaine d'application. En résumé, le recouvrement de la conception est la création de documents de conception utiles pour la maintenance d'un logiciel [13].

1.2 Le paradigme objet

Cette section a comme objectif de faire un bref rappel sur la technologie orientée objets et de définir la terminologie de base qui est utilisée en orienté objets [12]. Une façon de faire la réingénierie de programmes procéduraux est d'aller vers la technologie orientée objets.

L'idée de base

Le monde est plein d'éléments; il est possible d'abstraire ces éléments et d'appeler ces abstractions des objets. Par exemple, un chien, un chat et un cheval sont des objets qui peuvent être représentés par la classe animal. Le tableau 1.1 montre un exemple de la classe animal.

Terminologie de base

Objet Un objet est un abstraction d'un concept du domaine d'application. Un objet est caractérisé par des attributs et des méthodes pour agir sur ses attributs.

Attribut Un attribut permet de conserver un état pour l'objet d'un concept. Selon [6], l'attribut représente le domaine du problème et les responsabilités du système. Le domaine d'un attribut est l'ensemble des valeurs que celui-ci peut prendre. Dans le

ANIMAL

Nom	Espèce	Race	Sexe	Date de naissance	Poids
Jessy	Chien	Berger Allemand	Femelle	18-09-90	65 lbs
Minouche	Chat	Croisé	Femelle	14-04-92	13 lbs
Trésor	Cheval	Percheron	Mâle	05-06-84	1200 lbs
Naître					
Décéder					

TAB. 1.1: Exemple de la classe animal.

tableau 1.1, les attributs sont : `Nom`, `Espèce`, `Race`, `Sexe`, `Date de naissance` et `Poids`.

Méthode Une méthode permet de modifier la valeur d'un attribut. De fait, la méthode agit directement sur l'état de l'objet. L'ensemble des méthodes d'un objet permet de décrire le comportement de celui-ci. Dans le tableau 1.1, les méthodes sont `Naître` et `Décéder`.

Instance Une instance est un élément d'une classe. Dans la classe animal présentée au tableau 1.1, il y a trois instances. D'abord, il y a l'instance `Jessy`, `Chien`, `Berger Allemand`, etc. Ensuite, il y a celle débutant par `Minouche`, `Chat`, etc. Et finalement, il y a l'instance `Trésor`, `Cheval`, `Percheron`, etc.

Classe Une classe est le regroupement de plusieurs objets ayant des attributs et des méthodes communes. Dans l'exemple précédent, le tableau 1.1, la classe est `Animal`.

Héritage L'héritage est un mécanisme permettant d'exprimer le partage d'attributs entre des classes. Donc, l'héritage permet de simplifier la définition d'une classe similaire à une classe déjà définie.

Généralisation/Spécialisation Lorsque plusieurs classes se partagent des attributs, il est possible de créer une classe générale ayant comme attributs l'ensemble des attributs communs aux autres classes. On peut alors éliminer les attributs communs des autres classes. Ces classes deviennent des **spécialisations** de la classe générale. Par exemple, si la classe X est une spécialisation de la classe Y, alors toute instance de X est également une instance de Y. Le mécanisme inverse de la spécialisation est la généralisation, qui est l'action de créer une superclasse.

Aggrégation L'ensemble des attributs communs pour une classe sert en fait à définir la classe. L'**aggrégation** est le mécanisme qui permet de regrouper ces attributs pour former une classe.

Modularité La modularité exige que les objets soient le plus possible autonomes et cohésifs. La modularité est caractérisée par une forte cohésion et un couplage faible.

Encapsulation L'encapsulation est synonyme de masquage de l'information. L'encapsulation permet de ne pas montrer les détails. Contrairement à l'abstraction, le niveau de détail de l'information n'est pas connu.

1.3 La rétro-ingénierie des logiciels

Pour faire la réingénierie de programmes procéduraux vers des programmes orientés objets, il faut commencer par une phase de rétro-ingénierie. Cette phase a comme objectif d'identifier les éléments du code source procédural qui sont de bons candidats pour créer des objets. Par conséquent, la rétro-ingénierie est essentielle dans le processus de réingénierie d'un logiciel. Suivant Chikosfky et Cross [5], il y a sept objectifs que la rétro-ingénierie des logiciels permet d'atteindre. Le tableau 1.2 présente ces objectifs.

1- Réduire la complexité du logiciel
2- Générer des vues alternatives
3- Recouvrer les informations perdues
4- Détecter les effets de bord et analyser la qualité
5- Faciliter la réutilisation
6- Alimenter un dépôt ou une base de connaissances
7- Synthétiser un haut niveau d'abstraction

TAB. 1.2: Objectifs de la rétro-ingénierie des logiciels.

La rétro-ingénierie des logiciels peut être réalisée principalement par deux types d'approches. Il y a les approches classiques de rétro-ingénierie des logiciels et les approches utilisant les réseaux neuronaux. Les sous-sections qui suivent présentent quelques-unes de ces approches.

1.3.1 Les approches classiques

Les approches classiques de rétro-ingénierie des logiciels permettent d'atteindre les objectifs présentés au tableau 1.2. La première approche présentée concerne la réutilisation de composantes logicielles, tandis que la seconde approche joue un rôle dans la maintenance des logiciels.

1 - Le paradigme RE^2 — Réutilisation

La rétro-ingénierie des logiciels permet de détecter les composantes logicielles réutilisables. Le paradigme RE^2 permet la **ré-utilisation** et la **ré-ingénierie** de modules logiciels. C'est pour cette raison qu'il est appelé le paradigme RE^2. Ce modèle comporte les cinq phases décrites dans le tableau 1.3 [3].

1 - Candidature
2 - Élection des candidats
3 - Qualification
4 - Classification et emmagasinage
5 - Recherche et affichage

TAB. 1.3: Phases du paradigme RE^2.

1 - Candidature : cette phase regroupe les activités d'analyse du code source et elle produit des ensembles de composantes logicielles *possiblement* réutilisables. Chacun de ces ensembles est un candidat à la création d'un module réutilisable. Évidemment, pour que l'ensemble devienne un module réutilisable, il devra subir un certain traitement. Il faudra ajouter certaines composantes logicielles à cet ensemble, le réorganiser et probablement le généraliser. Voici les trois étapes de cette phase :

1.1 - Définir l'information à extraire et les critères de sélection

Cette étape permet de définir l'information à extraire du code source pour obtenir des composantes logicielles réutilisables. De plus, dans cette étape, on doit identifier les critères de sélection des composantes logicielles. Ces critères sont basés sur un ensemble de qualités et d'attributs représentant le potentiel de réutilisation d'une composante logicielle.

1.2 - Extraire les composantes logicielles

Cette étape permet d'extraire les composantes logicielles. L'extraction se réalise dans un processus de rétro-ingénierie des logiciels. Il s'agit de l'activité principale de la phase de candidature.

1.3 - Appliquer les critères de sélection

Cette étape permet de produire un ensemble de modules *possiblement* réutilisables. Les modules sont choisis par l'application des critères de sélection sur

l'information extraite. Un module *possiblement* réutilisable est un module qui est candidat pour être réutilisé. Un module qui est choisi pour être réutilisé sera traité dans les phases suivantes du paradigme RE^2.

2 - Élection des candidats : cette phase regroupe les activités d'analyse des modules candidats à la réutilisation et elle produit un ensemble de modules réutilisables. Voici les trois étapes de cette phase :

2.1 - Définir le gabarit d'un module

Cette étape permet de définir le gabarit utilisé pour regrouper les modules sélectionnés comme étant de bons candidats à la réutilisation. Donc, cette étape permet d'éliminer les candidats trop difficiles à réutiliser.

2.2 - Extraire les composantes logicielles de l'environnement externe

Cette étape permet d'extraire les composantes logicielles appartenant à l'environnement externe. Il s'agit d'un processus qui fait l'extraction des composantes ne faisant pas partie des modules candidats à la réutilisation. Ces composantes n'ont pas été retenues lors de la candidature, puisqu'il ne s'agit pas de modules entiers.

2.3 - Produire les modules réutilisables

Cette étape permet de regrouper l'ensemble des composantes logicielles pour en faire des modules réutilisables.

3 - Qualification : cette phase regroupe les activités destinées à produire les spécifications de chacun des modules réutilisables. La phase de qualification produit les spécifications fonctionnelles et les spécifications des interfaces. Voici les trois étapes de cette phase :

3.1 - Définir un formalisme de spécification

Cette étape permet de définir un formalisme de spécification. Ce formalisme exprime le fonctionnement du module réutilisable et indique comment ce module doit être utilisé.

3.2 - Produire les spécifications

Cette étape permet de produire les spécifications fonctionnelles et les spécifications des interfaces de chaque module réutilisable. Ces spécifications sont produites selon le formalisme défini à l'étape précédente. Cette étape peut être simplifiée en utilisant la documentation du système dont les composantes logicielles ont été extraites.

3.3 - Tests et corrections des spécifications produites

Cette étape permet de vérifier si les spécifications sont conformes au code source des modules réutilisables.

4 - Classification et emmagasinage : cette phase a pour objectif de classer les modules réutilisables et de créer le dépôt servant à conserver ces modules.

5 - Recherche et affichage : cette phase a pour objectif de permettre au programmeur d'interroger le dépôt pour récupérer un module réutilisable. Cette interrogation du dépôt doit être conviviale et les modules doivent être facilement accessibles.

2 - Un modèle algébrique — Maintenance

Les gens qui conçoivent des systèmes d'interrogation de code source font face à de gros problèmes. En effet, ils ont de la difficulté à trouver des modèles intégrés permettant de formuler simplement des requêtes sur le code source et de bien représenter les concepts extraits de ce dernier. L'utilisation d'un modèle basé sur l'algèbre relationnelle permettrait d'utiliser un langage formel de requête sur le code source.

Dans [16], on propose un modèle algébrique permettant de résoudre les problèmes mentionnés précédemment. En effet, le SCA (Source Code Algebra) est un modèle algébrique servant de base à la construction des systèmes de rétro-ingénierie des logiciels. L'algèbre permet de définir un modèle qui représente les informations dans le code source. En outre, elle permet de donner un ensemble d'opérateurs pouvant être utilisés pour formuler les requêtes sur le code source. L'utilisation de l'algèbre comme base d'un langage de requête peut être bénéfique. En effet, l'algèbre donne la capacité de produire des spécifications formelles lors de la construction d'un langage de requête et offre des opportunités concernant l'optimisation des requêtes.

Le modèle SCA permet de représenter des informations du code source et contient les opérateurs nécessaires à la formulation de requêtes sur les informations. Le modèle SCA voit le code source comme un domaine d'objets caractérisés par des attributs. Ces objets permettent d'emmagasiner les composantes d'information, les relations avec d'autres objets, les méthodes de calcul et toutes autres informations pertinentes. De plus, le modèle SCA supporte la notion de collection d'objets. Une collection peut être vue comme des ensembles ou comme des séquences d'objets. Finalement, les requêtes sont formulées en écrivant des expressions utilisant les opérateurs pouvant agir sur les objets individuels et sur leurs collections.

1.3.2 Les approches utilisant les réseaux neuronaux

L'étude des réseaux neuronaux a permis de constater qu'il est possible de les utiliser en rétro-ingénierie des logiciels. En effet, les réseaux neuronaux sont utiles pour augmenter la modularité des logiciels, ainsi que pour recouvrer leur conception à partir d'une analyse du code source.

La modularité des logiciels diminue les problèmes associés à la réutilisation, à la maintenance et à la restructuration du code source. Une technique permettant d'augmenter la modularité d'un logiciel est présentée dans [19]. Le réseau neuronal utilisé par cette

technique est un réseau qui fait la rétro-propagation de l'erreur. Arseneau et Spracklen appliquent une technique de réingénierie permettant de rendre la structure d'un logiciel plus modulaire [2]. Washburne et al. [22] offrent une méthode d'analyse permettant d'extraire et de classer les différents modules d'un code source.

Par ailleurs, le recouvrement de la conception des logiciels est un élément important dans la compréhension des logiciels. Une approche utilisant les réseaux neuronaux pour transformer un code procédural en un code orienté objets est proposée dans [1]. Finalement, Merlo et al. [15] suggèrent une méthode d'analyse du code source d'un logiciel qui est basée sur les commentaires et les identificateurs d'un programme.

Les sous-sections suivantes présentent davantage ces deux méthodes.

1 - Classement automatique des modules

Lorsqu'il est question de compréhension de logiciels, on pense à simplifier ceux-ci d'une manière quelconque. Cette simplification est un avantage lors de la maintenance et de la réutilisation d'un logiciel. Pour simplifier un logiciel, il est possible de classer ses modules en différentes catégories. La méthode de classement des modules d'un logiciel qui est présentée dans [22] utilise les réseaux neuronaux.

La méthode de classement des modules utilise sept superclasses. Chacune de ces superclasses est composée d'un ensemble de classes. Il y a de une à vingt-neuf classes par superclasse. Chacune des superclasses doit être classée à l'aide de son propre réseau neuronal. Évidemment, une superclasse ayant seulement une classe ne requiert pas de classement, car tous les modules se retrouveront dans cette seule classe. Chaque réseau doit avoir une architecture unique, car chaque superclasse a un nombre de classes différent. Chaque module d'un logiciel se retrouvera au moins dans une classe de chacune des superclasses.

Les réseaux neuronaux utilisés pour le classement des modules sont des réseaux neuronaux probabilistes à alimentation avant. Les réseaux neuronaux probabilistes sont une ex-

tension du modèle de Boltzman et des réseaux neuronaux multicouches. Habituellement, le nombre d'échantillons d'entraînement d'un réseau neuronnal devrait correspondre à dix fois le nombre de classes de la superclasse à laquelle appartient le réseau neuronal.

2 - Analyse des informations informelles

Le modèle qui est présenté dans cette section est tiré de [15]. Ce modèle utilise les informations informelles du code source pour recouvrer la conception du logiciel. Les informations informelles dans le code source sont les commentaires et les identificateurs. Un système qui est basé sur les réseaux neuronaux est capable de trouver les concepts appropriés en considérant l'information reçue en entrée. Cette caractéristique propre aux réseaux neuronaux se nomme la **mémoire associative**. Il s'agit de la capacité d'établir des associations entre des entrées et des sorties.

Étapes du processus d'analyse des informations informelles

1. Un expert humain doit :
   - définir les tables d'abréviations utilisées dans le code source;
   - procurer au système les lois de classification des concepts;
   - définir l'architecture du réseau neuronal.
2. Extraction des informations informelles.
3. Entraînement du réseau neuronal.
4. Génération des concepts correspondant aux informations informelles.

Un modèle utilisant un réseau neuronal est un modèle mathématique pouvant être vu comme un ensemble de neurones. Ces neurones sont reliées par des connexions ayant chacune un poids. Chaque neurone est activée par une entrée qui est reçue soit par une autre neurone, soit par le vecteur d'entrées du réseau. Le réseau utilisé par ce modèle suit un mode d'apprentissage supervisé.

1.4 L'identification des objets

Le code source d'un programme procédural contient des éléments qui peuvent faire penser à des objets [1]. On peut identifier et extraire ces éléments, cachés dans le code source, à l'aide d'approches de compréhension des programmes.

1.4.1 La compréhension des programmes

Une très grande partie de l'effort requis pour le développement d'un logiciel est consacré uniquement à sa maintenance. Par conséquent, une diminution du temps requis pour la maintenance d'un logiciel réduirait significativement son coût de développement. De plus, plusieurs études ont démontré qu'environ 50% de l'effort requis pour la maintenance d'un logiciel est consacré exclusivement à sa compréhension. À la lumière de ces observations, il apparaît évident que l'on doit se munir d'outils, de techniques, de représentations et d'approches qui nous aideront à comprendre les programmes.

D'une manière générale, il y a deux types d'approche de compréhension des programmes. Il y a l'approche ascendante, qui consiste à générer une description à partir du code source. Il y a l'approche descendante, qui consiste à formuler des hypothèses et à les confirmer en consultant le code source. Une approche qui se nomme "raffinement synchronisé" est présentée dans [18]. Cette approche est une combinaison de l'approche ascendante et de l'approche descendante.

Les représentations graphiques permettent de comprendre plus facilement les programmes. Lors de la compréhension des programmes, au lieu de dire qu'une image vaut mille mots, on peut dire qu'une représentation graphique vaut des milliers de lignes de code. La sous-section suivante présente des représentations graphiques aidant à la compréhension des programmes.

---

[1] Il s'agit de la traduction libre de *Object-Like Features*.

1.4.2 L'abstraction des programmes

L'abstraction en générale

En 1983, Wasserman définit l'abstraction comme étant la notion psychologique permettant de se concentrer sur un problème à un niveau de généralisation omettant les détails de bas niveau. De plus, il affirme que l'abstraction permet de travailler avec des concepts et des termes familiers à l'environnement du problème [17].

Le niveau d'abstraction d'un processus de rétro-ingénierie et les outils pouvant être utilisés pour supporter ce processus font référence aux informations de conception pouvant être extraites du code source. Idéalement, le niveau d'abstraction produit par ce processus doit être le plus haut possible. Le processus de rétro-ingénierie utilise le code source d'un programme pour en reproduire un haut niveau d'abstraction. Les représentations utiles pour l'identification des objets dans un code source procédural sont : le graphe de références, le graphe de visibilité des types et le graphe d'appels des fonctions.

Le graphe de références

Comme nous l'avons mentionné précédemment, le graphe de références d'un programme source est un graphe bipartite. Ce graphe bipartite a deux types de noeuds : des noeuds pour représenter les variables globales et des noeuds pour représenter les fonctions du programme. Les arcs entre les noeuds signifient qu'une fonction fait référence à une variable globale. Le graphe de références est une abstraction d'un programme source, car il permet de visualiser uniquement l'information pertinente à l'identification des objets. La figure 1.4 présente le graphe de référence du programme `Exemple1.cc`. Le code source de ce programme est disponible à l'annexe A.

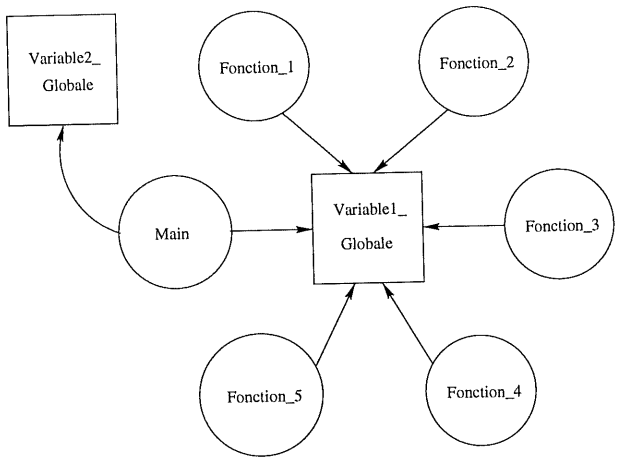

FIG. 1.4: Graphe de références du programme `Exemple1.cc`.

Le graphe de visibilité des types

Comme le graphe de références, le graphe de visibilité des types d'un programme source est un graphe bipartite. Les noeuds de ce graphe représente des fonctions et des types complexes de données. Un arc entre un noeud représentant une fonction et un noeud représentant un type complexe de données indique la présence d'une donnée de ce type complexe soit dans la déclaration, soit dans le corps de la fonction. Les types complexes de données peuvent être attribués à une variable globale, à une variable locale ou à un paramètre formel d'une fonction. La figure 1.5 présente le graphe de visibilité des types du programme `Exemple1.cc`.

Le graphe d'appels des fonctions

Le graphe d'appels des fonctions est un graphe ayant un seul type de noeuds. Chaque noeud représente une fonction et les arcs entre deux noeuds signifient qu'une fonction appelle une autre fonction. La figure 1.6 présente le graphe d'appels des fonctions du programme `Exemple1.cc`. Ce graphe permet de conserver la trace des fonctions n'apparaissant pas dans le graphe de références et dans le graphe de visibilité des types. Dans le graphe de références, il est possible d'observer uniquement les fonctions faisant réfé-

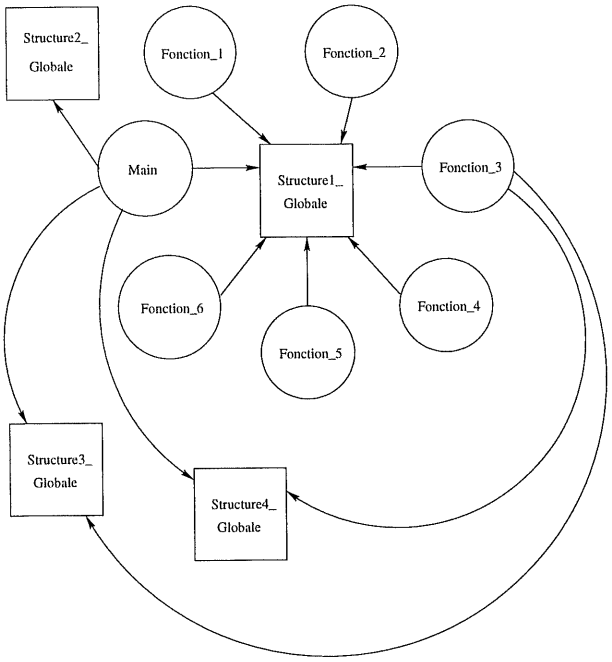

FIG. 1.5: Graphe de visibilité des types du programme `Exemple1.cc`.

rence à une variable globale. Par conséquent, toutes les autres fonctions sont oubliées. Lorsqu'on veut réécrire un programme procédural dans un langage orienté objets, il faut inclure toutes les fonctions dans le nouveau code source. De plus, il faut être capable de savoir à partir de quelles méthodes ces fonctions sont appelées.

1.4.3 La démarche d'identification des objets

La démarche générale

Comme nous l'avons signalé précédemment, il y a deux types d'approches permettant l'identification des objets dans un code source procédural. Il s'agit des approches dites dépendantes du domaine et des approches dites indépendantes du domaine. Tout d'abord, l'approche dite dépendante du domaine d'application exige des connaissances d'un expert pour définir un modèle de l'application. Ce modèle, utilisé avec le code source procédural de l'application, permet de recouvrer la conception pour ensuite être capable

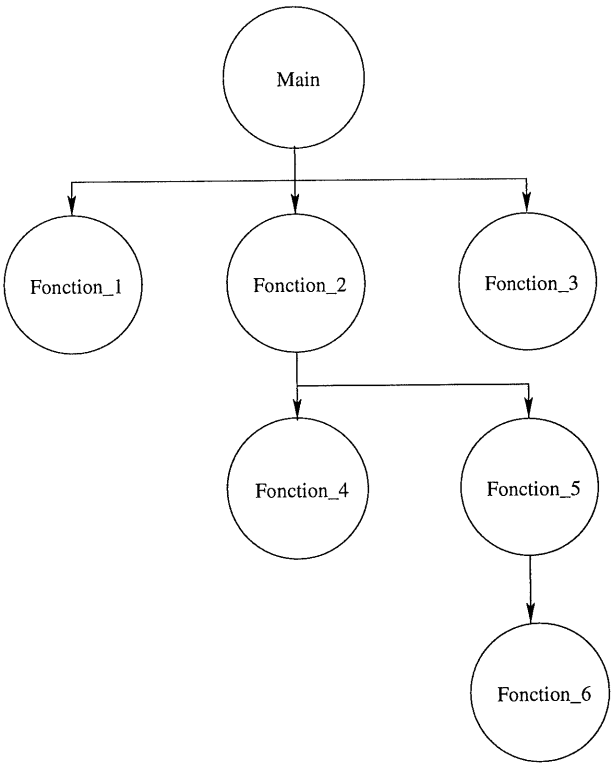

FIG. 1.6: Graphe d'appels des fonctions du programme `Exemple1.cc`.

d'identifier les objets de cette application. Ensuite, il y a l'approche dite indépendante du domaine de l'application. Cette approche ne demande aucune expertise du domaine de l'application et utilise uniquement le code source de l'application pour identifier les objets. Par conséquent, le niveau d'automatisation du processus permettant l'identification des objets est beaucoup plus élevé avec l'approche indépendante du domaine qu'avec l'approche dépendante du domaine. Les travaux sont présentés en utilisant les mêmes critères de présentation pour chacune des approches. Cela a comme objectif de permettre une comparaison des approches.

Les critères de présentation

Il y a cinq critères de présentation dont il faut tenir compte pour bien comprendre ces approches :

1. l'information utilisée en entrée (code source, modèles d'analyse, expert, etc.);
2. les abstractions utilisées;
3. les étapes de l'approche;
4. la portée de l'approche (identification des objets ou génération de code source);
5. la classification de l'approche (dépendante vs. indépendante du domaine, théorie des graphes, connaissances, heuristiques, etc.).

Les sous-sections qui suivent présentent des travaux réalisés en utilisant ces deux types d'approches.

Les approches dépendantes du domaine

La première approche présentée est celle de Gall et al. [7], [8], [9] et [10]. Cette approche est dépendante du domaine de l'application, car elle nécessiste un modèle d'analyse de l'application. Il faut débuter par l'extraction du graphe d'appels de l'application. Ensuite, le graphe d'appels permet d'extraire le diagramme de flux de données (DFD)

de l'application. Le DFD est utilisé pour extraire le diagramme entitées-relations (DER). Le DER permet la création du modèle des classes. Lorsque l'extraction des graphes est terminée, il faut créer le *Reverse object-oriented Application Model* (RooAM). Le RooAM est un modèle de conception orienté objets créé à l'aide du modèle des classes. Aussi, il faut créer le *Forward object-oriented Application Model* (FooAM). Le FooAM est un modèle de conception orienté objets créé à l'aide du modèle d'analyse de l'application. Ensuite, il faut faire un *mapping* entre le RooAM et le FooAM pour obtenir le modèle orienté objets de l'application. Le modèle orienté objets de l'application est transformé en code source orienté objets. Lors de la mise en application de cette approche, il est possible de rencontrer des cas d'exception ne pouvant pas être traités à l'aide de ce modèle. Ces cas d'exception doivent être traités manuellement lors de la transformation du code source. Le tableau 1.4 résume cette approche.

La seconde approche présentée est celle de Shin [20]. Cette approche est dépendante du domaine et elle est inspirée de l'approche précédente. Contrairement à l'approche de Gall et al., cette approche utilise le graphe de références comme abstraction. D'abord, il faut extraire le graphe de références de l'application. Ensuite, il faut identifier les objets en se basant sur les variables globales de l'application. Lorsque les objets sont identifiés, il faut identifier les méthodes de chaque objet à l'aide du graphe de références et du code source. Cette approche se termine par la création des classes d'objets et la réorganisation de code source procédural en un code source orienté objets. Soulignons que le déroulement de cette approche est supporté par un navigateur. Ce navigateur permet d'effectuer des requêtes sur le code source pour la prise de décisions. Le tableau 1.5 résume cette approche.

Les approches indépendantes du domaine

La troisième approche présentée est celle de Canfora et al. [4]. Cette approche présente un meilleur niveau d'automatisation que les approches précédentes, car elle est

indépendante du domaine de l'application. De plus, les informations requises en entrée sont le code source et des heuristiques statistiques. D'abord, il faut extraire le graphe de références de l'application. Ensuite, il faut calculer l'indice de connectivité entre chaque noeud du graphe, ainsi que la variation de cet indice de connectivité. Ces informations sont utiles à la résolution des conflits qui surviennent lors de l'identification des objets dans le code source procédural. En effet, ces indices permettent de déterminer si on doit découper ou regrouper les objets. Le tableau 1.6 résume cette approche.

La dernière approche présentée est celle de Yeh et al. [25]. L'utilisation du graphe de visibilité des types permet d'identifier des objets dans un code source procédural même si celui-ci ne possède pas de variables globales. Il faut débuter par l'extraction du graphe de références et du graphe de visibilité des types. Ensuite, il faut faire l'union de ces deux graphes en ne dupliquant pas les éléments. L'union des deux graphes permet d'identifier les objets, ainsi que les types abstraits de données (TAD) par la technique des composantes connexes. Le tableau 1.7 résume cette approche.

APPROCHE 1

Critères	
1-	Le code source de l'application Le modèle d'analyse de l'application Un expert du domaine de l'application
2-	Le graphe d'appels Le diagramme de flux de données (DFD) Le diagramme entités-relations (DER) Un modèle de classes
3-	Étapes : 3.1- L'extraction des abstractions utilisées 3.2- La création du *Reverse object-oriented Application Model* (RooAM) 3.3- La création du *Forward object-oriented Application Model* (FooAM) 3.4- Le *mapping* entre le RooAM et le FooAM 3.5- La réorganisation du code source et le traitement des exceptions
4-	La génération du code source
5-	L'approche est dite dépendante du domaine L'approche requiert des connaissances d'un expert

TAB. 1.4: Approche de Gall et al.

APPROCHE 2

Critères	
1-	Le code source de l'application Le modèle d'analyse de l'application Un expert du domaine de l'application
2-	Le graphe de références Un navigateur
3-	Étapes : 3.1- L'extraction du graphe de références 3.2- L'identification des classes d'objets 3.3- L'identification des méthodes 3.4- La réorganisation du code source
4-	La génération du code source
5-	L'approche est dite dépendante du domaine L'approche requiert des connaissances d'un expert

TAB. 1.5: Approche de Shin.

APPROCHE 3

Critères	
1-	Le code source de l'application Des heuristiques statistiques
2-	Le graphe de références
3-	Étapes : 3.1- L'extraction du graphe de références 3.2- Le calcul de l'indice de connectivité 3.3- Le calcul de la variation de l'indice de connectivité 3.4- La résolution des conflits
4-	L'identification des objets
5-	L'approche est dite indépendante du domaine L'approche utilise la théorie des graphes L'approche utilise des heuristiques

TAB. 1.6: Approche de Canfora et al.

APPROCHE 4

Critères	
1-	Le code source de l'application
2-	Le graphe de références Le graphe de visibilité
3-	Étapes : 3.1- L'extraction des abstractions utilisées 3.2- L'union du graphe de références et du graphe de visibilité 3.3- L'identification des objets et des type abstraits de données (TAD) 3.4- La résolution des cas de composantes mixtes
4-	L'identification des objets
5-	L'approche est dite indépendante du domaine L'approche utilise la théorie des graphes

TAB. 1.7: Approche de Yeh et al.

Chapitre 2

LA SOLUTION

La solution proposée dans ce mémoire est inspirée des travaux de Canfora et al. [4]. L'approche qu'ils suggèrent permet d'identifier et de détruire les connexions indésirables dans un graphe de références. L'identification et la destruction de ces connexions indésirables permettent de produire des regroupements de variables globales et de fonctions. Ces regroupements sont des candidats pour devenir des objets. En effet, dans un regroupement, les variables globales représentent les données de l'objet, tandis que les fonctions représentent les méthodes de cet objet. En premier lieu, ce chapitre présente l'algorithme d'identification des objets tel que développé par Canfora et al. [4]. En deuxième lieu, il expose les notions théoriques nécessaires à la focalisation d'une fonction. En troisième lieu, il présente les problèmes que nous avons identifiés avec l'approche de Canfora et al. [4]. Finalement, ce chapitre se termine par la présentation de nos contributions face aux problèmes identifiés avec l'approche de Canfora et al. [4].

2.1 L'algorithme d'identification des objets

Cette section présente l'algorithme permettant l'identification des objets dans un code source procédural [4]. Cet algorithme d'identification d'objets permet d'éliminer les connexions indésirables dans un graphe de références. Comme nous l'avons mentionné

précédemment, le graphe de références d'un programme procédural est construit à partir des variables globales et des fonctions de celui-ci. L'algorithme permet de produire des regroupements de fonctions et de variables globales. Les variables globales permettent d'emmagasiner l'état de l'objet, tandis que les fonctions implantent les méthodes de cet objet. Les regroupements produits par l'algorithme sont des candidats pour devenir des objets.

2.1.1 Le graphe de références

L'algorithme d'identification des objets se termine lorsque le graphe de références est transformé en un ensemble de sous-graphes isolés. Chacun de ces sous-graphes est un candidat pour devenir un objet. Comme nous l'avons vu plus haut, l'algorithme utilise en entrée le graphe de références : c'est un graphe bipartite composé de deux ensembles de noeuds et d'un ensemble d'arcs. Voici les deux ensembles dont il faut tenir compte pour créer les noeuds du graphe de références :

- **VARIABLES_GLOBALES** : l'ensemble des variables globales;
- **FONCTIONS** : l'ensemble des fonctions.

Un arc représente l'utilisation d'une variable globale par une fonction. Les noeuds et les arcs du graphe de références sont décrits formellement par les deux ensembles ci-dessous :

- **NOEUDS** : $VARIABLES\_GLOBALES \bigcup FONCTIONS$;
- **ARCS** : $\{(f, d) \mid f \in FONCTIONS \wedge d \in VARIABLES\_GLOBALES \wedge f$ *accède à* $d\}$.

Si le programme procédural était écrit avec une approche pleinement orientée objets, les objets du graphe seraient associés à des sous-graphes connexes. Malheureusement, les programmes procéduraux ne sont généralement pas écrits avec une telle approche. Certaines fonctions accèdent à plusieurs variables. Lorsque ces variables appartiennent à des objets différents, cela crée des connexions indésirables entre les sous-graphes. Il y

a deux types de connexions indésirables entre les sous-graphes : les connexions résultant de la coïncidence et les connexions résultant d'une mauvaise conception.

Connexions par coïncidence : ces connexions résultent de fonctions qui implantent plus d'une fonctionnalité. En d'autres mots, les connexions par coïncidence sont la conséquence de fonctions qui implantent plus d'une méthode avec au moins deux de ces méthodes qui appartiennent à des objets différents. Par exemple, une fonction qui initialise des variables globales peut produire ce genre de connexions. Dans un programme procédural, il est normal d'avoir une fonction pour initialiser les variables globales. Par contre, dans un programme orienté objets, cela n'est pas souhaitable, car chaque objet possède son constructeur pour initialiser ses variables.

Connexions par mauvaise conception : ces connexions résultent de fonctions qui accèdent les données de plus d'un objet dans le but d'implanter des opérations spécifiques au système. Habituellement, ces connexions sont le résultat d'un mélange entre les méthodes de conception basées sur la décomposition fonctionnelle et l'approche orientée objets. Généralement, il s'agit de fonctions ayant un couplage élevé et une faible cohésion.

La meilleure façon d'éliminer les connexions indésirables consiste à appliquer une technique de focalisation [1] pour séparer les fonctions qui génèrent ces connexions. Par exemple, une fonction qui initialise les variables globales appartenant à trois objets peut être séparée en trois sous-fonctions. Ces trois sous-fonctions vont devenir des méthodes, c'est-à-dire des constructeurs de chacun des objets.

L'algorithme d'identification des objets tente d'identifier les sous-graphes à l'aide d'un processus itératif. À chaque itération, on associe à chaque fonction f du système un indice

[1] La focalisation consiste à extraire le code source concernant une variable.

qui mesure la variation de la connectivité interne de son sous-graphe. Cette variation de la connectivité interne permet de générer de nouveaux regroupements. Le sous-graphe de la fonction f doit inclure toutes les données référencées par cette fonction et toutes les fonctions faisant référence uniquement à ces données. Le sous-graphe de la fonction f est défini par l'ensemble $V(f)$ et par l'ensemble $F(f)$. La sous-section suivante présente les éléments nécessaires à l'application de l'algorithme.

2.1.2 Les éléments nécessaires à l'application de l'algorithme

Voici les éléments qu'il faut considérer lors de l'application de l'algorithme d'identification des objets présenté à la section 2.1.4.

$GRAPHE\_REFERENCES(NOEUDS, ARCS)$ = le graphe de références

$PreSet(noeud) = \{y \mid y \in NOEUDS \wedge noeud \in NOEUDS \wedge (y, noeud) \in ARCS\}$

$PreSet(noeud) = \emptyset$ pour tout $noeud \in FONCTIONS$

$PostSet(noeud) = \{y \mid y \in NOEUDS \wedge noeud \in NOEUDS \wedge (noeud, y) \in ARCS\}$

$PostSet(noeud) = \emptyset$ pour tout $noeud \in VARIABLES\_GLOBALES$

$V(f)$ = l'ensemble des variables appartenant au sous-graphe de f

$$V(f) = PostSet(f)$$

$F(f)$ = l'ensemble des fonctions appartenant au sous-graphe de f

$$F(f) = \bigcup_{d \in PostSet(f)} \{f_i | f_i \in PreSet(d) \wedge PostSet(f_i) \subseteq PostSet(f)\}$$

$\# =$ la cardinalité d'un ensemble

$IC(f) =$ *l'indice de connectivité interne* du sous-graphe de f est le nombre d'arcs appartenant à des fonctions qui ne pointent pas à l'extérieur du sous-graphe de f divisé par le nombre d'arcs total appartenant à des variables du sous-graphe de f. $IC(f)$ permet de savoir si le sous-graphe de f est fortement connexe. Donc, plus $IC(f)$ est près de un, plus le sous-graphe de f est connexe.

$$IC(f) = \frac{\sum_{d \in PostSet(f)} \#\{f_i | f_i \in PreSet(d) \wedge PostSet(f_i) \subseteq PostSet(f)\}}{\sum_{d \in PostSet(f)} \#PreSet(d)}$$

$\Delta IC(f) =$ *la variation de l'indice de connectivité interne* du sous-graphe de f est $IC(f)$ moins la somme des ratios, pour chaque variable d pointée par f, entre le nombre de fonctions pointant **uniquement** d et le nombre total de fonctions pointant d. $\Delta IC(f)$ permet de déterminer si le sous-graphe de f est dans une forme minimale. Le sous-graphe de f est dans une forme minimale si le $\Delta IC(f) = 0$. Si le sous-graphe de f n'est pas dans une forme minimale, alors soit que f doit être focalisée, soit que les variables pointées par f doivent être regroupées.

$$\Delta IC(f) = IC(f) - \sum_{d \in PostSet(f)} \frac{\#\{f_i \mid PostSet(f_i) = \{d\}\}}{\#PreSet(d)}$$

$REGROUPER = \{f \mid \Delta IC(f) > seuil\}$

$FOCALISER = \{f \mid \Delta IC(f) < 0 \vee \Delta IC(f) > 0 \wedge \Delta IC(f) <= seuil\}$. Cette définition de l'ensemble $FOCALISER$ est le fruit d'une petite modification que nous avons apportée à la définition de l'ensemble $FOCALISER$ de Canfora et al. [4]. En effet, la définition originale ne permet pas de focaliser les fonctions ayant un $\Delta(IC)$ négatif. Cette modification est justifiée puisqu'elle permet d'obtenir les mêmes résultats que Canfora et al. [4].

2.1.3 Le seuil

Le seuil est très important, car il permet de déterminer si le sous-graphe associé à une fonction doit être modifié ou non. De plus, si le sous-graphe d'une fonction doit subir une modification, alors le seuil permet de déterminer le type de modification que ce sous-graphe devra subir. Canfora et al. [4] n'indiquent pas clairement leur manière de calculer le seuil lors de leurs expérimentations. Donc, lors de nos expérimentations, nous avons développé une approche pour le calcul du seuil. Cette approche nous permet d'obtenir le plus grand nombre de regroupements possibles et les résultats qu'elle donne sont significatifs. À la sous-section 2.3.3 nous présentons la manière que Canfora et al. [4] utilise pour calculer le seuil, tandis qu'à la sous-section 2.4.4 nous présentons notre manière de calculer le seuil.

2.1.4 L'algorithme

Pour chaque noeud $f \in FONCTIONS$ il faut calculer $IC(f)$, l'indice de connectivité du sous-graphe de f. Le sous-graphe de f est défini par l'ensemble des variables pointées par f, c'est-à-dire $V(f)$, et par l'ensemble des fonctions pointant uniquement les variables pointées par f, c'est-à-dire $F(f)$. Ensuite, pour chaque noeud $f \in FONCTIONS$, il faut calculer la variation de l'indice de connectivité, c'est-à-dire $\Delta IC(f)$. Les fonctions ayant une variation de l'indice de connectivité qui est suffisamment élevée sont utilisées pour générer des regroupements de variables. Par ailleurs, les fonctions ayant une variation de l'indice de connectivité qui n'est pas suffisamment élevée et qui est différente de zéro sont considérées comme des fonctions introduisant des connexions indésirables devant être focalisées. Évidemment, la variation de l'indice de connectivité de chaque fonction est recalculée à chaque itération de l'algorithme. L'algorithme termine lorsque chaque fonction accède à une et seulement une variable, c'est-à-dire lorsque le graphe de références est sous la forme d'un ensemble de sous-graphes isolés. Chaque sous-graphe est formé d'un noeud appartenant à l'ensemble VARIABLES_GLOBALES et d'un ou

plusieurs noeuds appartenant à l'ensemble FONCTIONS. Le noeud appartenant à VARIABLES_GLOBALES est possiblement associé à un groupe de données du système original, tandis que les noeuds appartenant à FONCTIONS pointent sur un noeud appartenant à VARIABLES_GLOBALES. Chacun des sous-graphes obtenus par cet algorithme est un candidat pour devenir un objet.

Le pseudo-code de l'algorithme

TANT-QUE le graphe de références n'est pas sous une forme d'un ensemble de sous-graphes isolés, c'est-à-dire que chaque sous-graphe est formé d'un noeud $\in VARIABLE\_GLOBALES$ et d'un ou plusieurs noeuds $\in FONCTIONS$.

FAIRE

POUR-CHAQUE noeud $f \in FONCTIONS$ **FAIRE**

- Calculer la valeur de $IC(f)$ et de $\Delta IC(f)$

FIN-POUR

- Calculer le seuil
- Calculer l'ensemble REGROUPER
- Calculer l'ensemble FOCALISER
- Interagir avec un expert pour effectuer certaines opérations sur le graphe (notre implantation de cet algorithme ne permet pas cette interaction) :
  - Détruire des fonctions du graphe;
  - Déplacer des fonctions de l'ensemble REGROUPER vers l'ensemble FOCALISER;
  - Déplacer des fonctions de l'ensemble FOCALISER vers l'ensemble REGROUPER.

POUR-CHAQUE noeud $f \in REGROUPER$ **FAIRE**

- Regrouper les variables pointées par les f en une seule variable
- Faire pointer les f sur la nouvelle variable

FIN-POUR

POUR-CHAQUE noeud $f \in FOCALISER$ **FAIRE**

- Déterminer les f qui sont réellement à focaliser, c'est-à-dire que la fonction f peut être focalisée si et seulement si elle pointe sur une variable regroupée et qu'elle pointe sur un autre ensemble de variables
- Trancher en deux les fonctions f qui sont réellement à focaliser. f_a pointe sur la variable regroupée, tandis que f_b pointe sur l'autre ensemble de variables

FIN-POUR

FIN-TANT-QUE

2.2 La focalisation

Selon Menif [14] et Weiser [23], la focalisation est un ensemble de techniques permettant à l'analyste de localiser et de comprendre les fonctionnalités d'un système. En 1984, Weiser [23] a introduit la notion de focalisation. Il s'agit d'une méthode de décomposition automatique des modules qui se base sur l'analyse statique du flot de données et du flot de contrôle. La focalisation permet de réduire le programme dans une forme minimale pour un comportement donné. Cette forme réduite d'un programme est une tranche. Des études ont démontré que des programmeurs d'expérience font mentalement une focalisation informelle lorsqu'ils corrigent des erreurs dans un programme. La focalisation est utilisée dans plusieurs domaines d'application. En outre, lors de la réingénierie des applications, la focalisation peut être utile pour la compréhension, la modularisation et la réutilisation des programmes.

La focalisation peut être réalisée à deux niveaux : le niveau **intraprocédural** et le niveau **interprocédural**. Le niveau intraprocédural concerne chacune des instructions d'une procédure, tandis que le niveau interprocédural concerne les appels de procédures.

Un critère C est un tuple $< s, V >$, où s est une instruction du programme et V est un sous-ensemble de variables. On peut définir une tranche S_C d'un programme sous un critère $C =< s, V >$. Cette tranche est une portion exécutable d'un programme formée de l'ensemble des instructions qui contribuent au calcul de V, **avant** l'exécution de l'instruction s. Par exemple, considérons la procédure `AFFICHER_TOTAL()`.

```
AFFICHER_TOTAL()
1-   BEGIN
2-       READ(X,Y)
3-       TOTAL := 0.0
4-       SUM   := 0.0
5-       IF X <= 1
6-          THEN  SUM := Y
7-          ELSE  BEGIN
8-             READ(Z)
9-             TOTAL  := X*Y
10-         END
11-      WRITE(TOTAL,SUM)
12-  END.
```

Il est possible d'extraire des tranches de cette procédure selon différents critères. Voici le résultat de l'application de l'algorithme sur l'exemple précédent.

```
C = <12,{Z}>
1-   BEGIN
2-      READ(X,Y)
3-      IF X <= 1
4-         THEN
5-      ELSE  READ(Z)
6-      END
7-   END.
```

Graphe Def/Use intraprocédural

L'algorithme de focalisation utilise le graphe *Def/Use*. Il s'agit d'un graphe de contrôle orienté possédant un noeud initial et un noeud final. Le noeud initial correspond au point d'entrée dans la procédure, tandis que le noeud final représente le point de sortie de la procédure. Les noeuds du graphe *Def/Use* intraprocédural représentent des instructions élémentaires d'une procédure, tandis que les arcs représentent le flot de contrôle entre ces instructions. Chaque noeud est annoté d'un ensemble de variables utilisées et d'un ensemble de variables définies. Une variable est dite *définie* dans une instruction si cette instruction affecte une valeur à la variable. Par ailleurs, une variable est dite *utilisée* dans une instruction si celle-ci doit évaluer la valeur de cette variable. Par conséquent, il est possible qu'une variable appartienne à l'ensemble des variables *définies* et à l'ensemble des variables *utilisées* d'une même instruction. Par exemple, dans l'instruction `X = X + Y`, la variable `X` appartient à l'ensemble des variables *définies* et à l'ensemble des variables *utilisées*, tandis que la variable `Y` appartient uniquement à l'ensemble des variables *utilisées* de cette instruction. La figure 2.1 représente le graphe *Def/Use* intraprocédural de la procédure `AFFICHER_TOTAL()`.

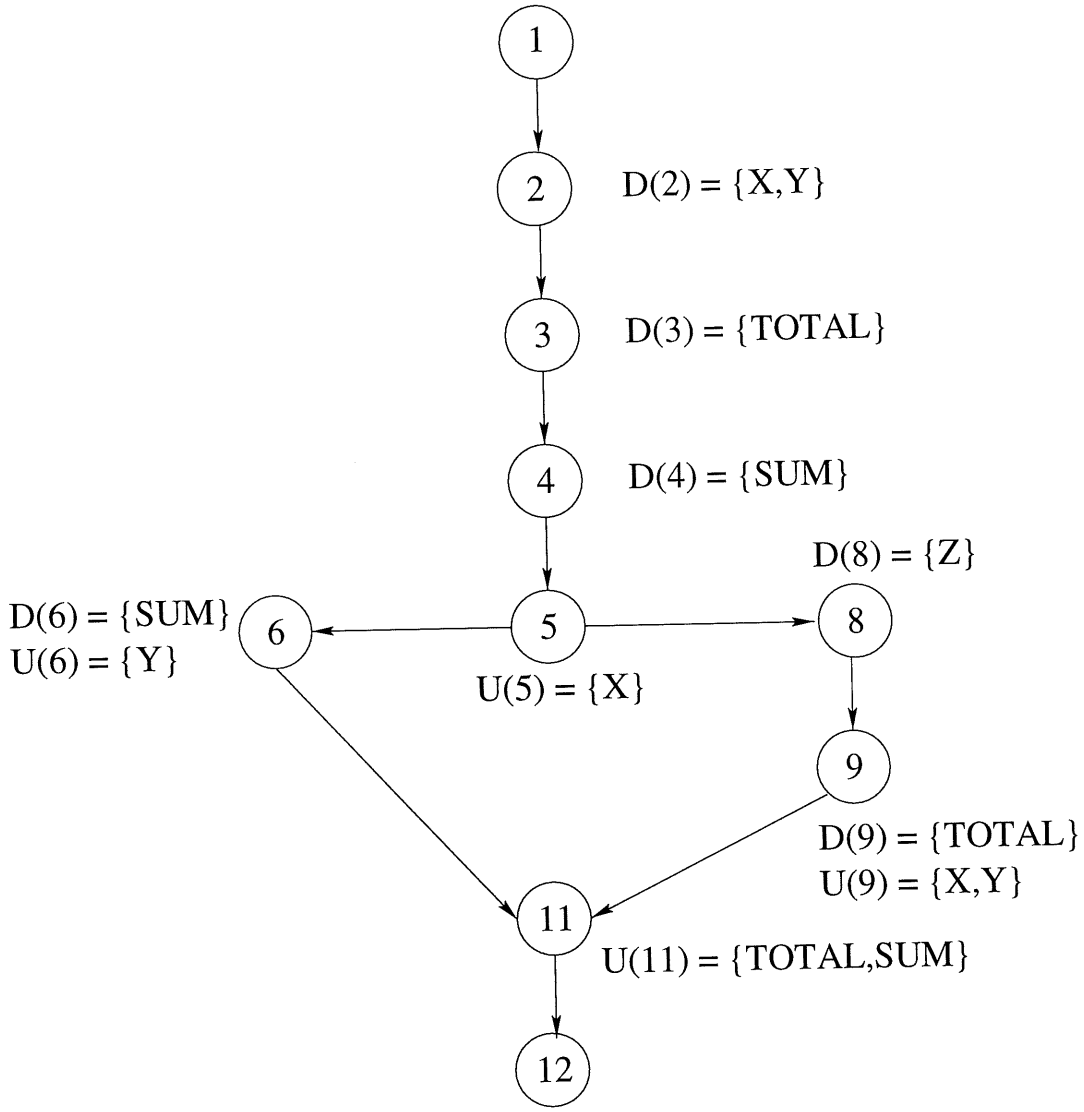

FIG. 2.1: Exemple d'un graphe *Def/Use* intraprocédural.

2.2.1 La focalisation de base

Selon Weiser [23], la focalisation de base a été utilisée essentiellement pour la compréhension des programmes. L'objectif de la focalisation de base est de trouver une tranche minimale, c'est-à-dire une tranche ayant un minimum d'instructions. Pour extraire cette tranche, il faut utiliser le graphe *Def/Use* et un algorithme récursif. La focalisation de base intraprocédural permet d'extraire une tranche S_C d'instructions d'une procédure affectant directement ou indirectement le calcul de V **avant** l'exécution de l'instruction s.

2.3 Les problèmes identifiés avec l'approche

Bien que l'approche présentée par Canfora et al. [4] puisse produire des regroupements intéressants, nous avons identifié quatre principaux problèmes lors de son utilisation. Premièrement, cette approche ne permet pas d'identifier les classes d'objets. Deuxièmement, elle ne produit pas de regroupements intéressants si le nombre de variables globales par rapport au nombre de fonctions n'est pas significatif. Troisièmement, le calcul du seuil est imprécis. Et finalement, la focalisation d'une fonction qui touche à deux sous-graphes est réalisée d'une manière systématique.

2.3.1 L'identification des classes d'objets

Lors de l'identification des objets dans un code source procédural, il peut s'avérer très utile de connaître la classe à laquelle appartient chaque objet. Comme nous l'avons mentionné précédemment, l'objet est une instance de la classe. Dans les programmes procéduraux, il est fréquent de retrouver une seule instance de chaque classe. Aussi, chaque classe possède un seul objet et l'identification des objets peut s'avérer une solution adéquate. Par contre, certains programmes procéduraux peuvent avoir plusieurs objets

appartenant à la même classe. Donc, l'identification des objets dans un code source procédural ne permettra pas d'identifier ces regroupements d'objets, c'est-à-dire les classes d'objets.

2.3.2 L'utilisation des variables globales

L'approche présentée par Canfora et al. [4] utilise le graphe de références comme entrée à l'algorithme d'identification des objets dans un code source procédural. L'utilisation du graphe de références peut s'avérer inutile si le nombre de variables globales par rapport au nombre de fonctions n'est pas représentatif du nombre de données par rapport au nombre de fonctions. Cette situation se produit lorsque le transfert des données entre les fonctions s'effectue principalement par paramètres.

2.3.3 Le calcul du seuil

Le seuil est déterminé par une approche statistique. Pour l'établir, il faut considérer la distribution de la variation de l'indice de connectivité. Le type de la distribution et le seuil sont établis selon les caractéristiques du système sur lequel l'algorithme est appliqué. En effet, le style et les standards de programmation qui ont été suivis influencent la distribution. Ces caractéristiques sont subjectives et difficilement identifiables. Donc, Canfora et al. [4] ne calculent pas le seuil d'une manière automatique.

2.3.4 La focalisation systématique

L'algorithme présenté par Canfora et al. [4] focalise une fonction qui touche à deux sous-graphes d'une manière systématique. Sachant que la focalisation n'est pas toujours possible et qu'elle est très coûteuse à réaliser, il est préférable de la minimiser.

2.4 Nos contributions

Pour pallier aux problèmes que nous avons présentés à la section précédente, nous avons développé un nouveau type de graphe : le graphe de visibilité des données. De plus, nous avons amélioré le graphe de références et le graphe de visibilité des types. Ces graphes sont utilisés en entrée de l'algorithme d'identification des objets présenté à la section 2.1.4. Le graphe de visibilité des types peut permettre d'identifier les classes d'objets d'un code source procédural. En outre, le graphe de visibilité des données permet de considérer toutes les données globales du code source procédural, c'est-à-dire les données globales déclarées globalement et les données globales passées en paramètre. Le graphe de références amélioré permet de ne pas focaliser une fonction qui touche à deux sous-graphes d'une manière systématique. Cette section présente notre manière de calculer le seuil, ainsi que notre modification de la condition de fin de l'algorithme présenté par Canfora et al. [4].

2.4.1 Le graphe de visibilité des types

Le graphe de visibilité des types est un graphe bipartite. Un premier type de noeuds représente les types complexes de données et un second type de noeuds représente les fonctions. Voici les deux ensembles dont il faut tenir compte pour créer les noeuds du graphe de visibilité des types :

- **TYPES_COMPLEXES_DONNEES** : l'ensemble des types complexes de données;
- **FONCTIONS** : l'ensemble des fonctions.

Un arc entre une fonction f et un type t indique la présence d'une donnée de type t soit dans les paramètres, soit dans le corps de la fonction f. Les noeuds et les arcs du graphe de visibilité des types sont décrits formellement par les deux ensembles ci-dessous :

- **NOEUDS** : $TYPES\_COMPLEXES\_DONNEES \bigcup FONCTIONS$;
- **ARCS** : $\{(f,t) \mid f \in FONCTIONS \wedge t \in TYPES\_COMPLEXES\_DONNEES \wedge f\ voit\ t\}$.

Notre contribution concernant le graphe de visibilité des types consiste en l'ajout d'un attribut sur les arcs. Cet attribut concerne la portée de la déclaration de la variable de type complexe de données. En effet, les types complexes de données peuvent être attribués à une variable globale, à une variable locale ou à un paramètre formel d'une fonction. Par conséquent, les valeurs possibles pour les attributs sont : $GLOBALE$, $LOCALE$ et $PARAMETRE$. Cet attribut est utilisé par l'expert pour évaluer l'exactitude des classes d'objets identifiées par l'algorithme. De fait, il peut être utile à l'expert de savoir qu'une variable de type complexe de données a été déclarée globale au lieu d'être déclarée locale à une fonction. La figure 1.5 présente le graphe de visibilité des types du programme `Exemple1.cc`.

Les regroupements produits par l'algorithme d'identification des objets présenté à la section 2.1.4, ne sont plus des regroupements de variables et de fonctions. Il s'agit de regroupements de types complexes de données et de fonctions. Ces regroupements sont des candidats pour devenir des classes d'objets. En effet, chaque type complexe de données est formé de types élémentaires de données. Par exemple, dans le programme `Exemple1.cc` de l'annexe A, on retrouve le type complexe de données `Structure1_Globale`.

```
typedef struct
    {
      int  Nombre1;
      char Lettre1;
    } Structure1_Globale;
```

Nous avons appliqué l'algorithme d'identification des objets au graphe de visibilité des types du programme `Exemple1.cc` dans le but d'identifier les classes d'objets. L'algorithme propose de regrouper les quatre types complexes de données, à savoir `Structu-`

re1_Globale, Structure2_Globale, Structure3_Globale et Structure4_Globale. Le nouveau type complexe de données résultant de ce regroupement se nomme Structure1_Globale_REGROUPER.

```
typedef struct
    {
      int  Nombre1;
      int  Nombre2;
      int  Nombre3;
      int  Nombre4;
      char Lettre1;
      char Lettre2;
      char Lettre3;
      char Lettre4;
    } Structure1_Globale_REGROUPER;
```

De plus, toutes les fonctions du graphe pointent sur ce nouveau type complexe de données. La figure 2.2 présente le regroupement produit par l'application de l'algorithme d'identification des objets sur le graphe de visibilité des types du programme Exemple1.cc.

L'intervention de l'expert pour obtenir des meilleurs résultats

Le regroupement obtenu par l'application de l'algorithme sur le graphe de visibilité des types du programme Exemple1.cc peut sembler exact. Par contre, un expert du programme Exemple1.cc aurait pu suggérer des regroupements plus appropriés. Il aurait pu par exemple proposer deux regroupements de types complexes de données, à savoir le regroupement Structure1_Globale_REGROUPER et le regroupement Structure3_Globale_REGROUPER.

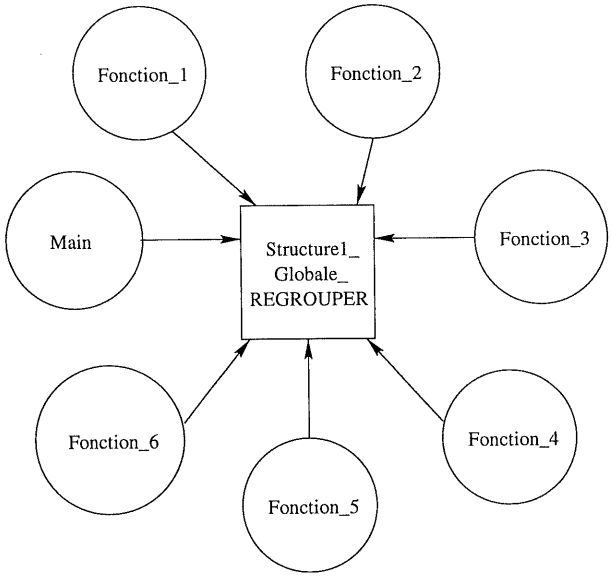

FIG. 2.2: Le graphe de visibilité des types après l'application de l'algorithme.

```
typedef struct
    {
      int  Nombre1;
      int  Nombre2;
      char Lettre1;
      char Lettre2;
    } Structure1_Globale_REGROUPER;

typedef struct
    {
      int  Nombre3;
      int  Nombre4;
      char Lettre3;
      char Lettre4;
    } Structure3_Globale_REGROUPER;
```

Les deux regroupements proposés par l'expert sont des candidats pour devenir des classes d'objets. Le premier candidat pour devenir une classe d'objets est le regroupement Structure1_Globale_REGROUPER. Ce candidat possède les attributs : Nombre1, Nombre2, Lettre1 et Lettre2. De plus, ce candidat possède les méthodes : Main_A, Fonction_1, Fonction_2, Fonction_3_A, Fonction_4, Fonction_5 et Fonction_6.

```
class Structure1_Globale_REGROUPER
    {
      public :
           int     Main_A();
           void    Fonction_1();
           void    Fonction_2();
           void    Fonction_3_A();
           void    Fonction_4();
           void    Fonction_5();
           void    Fonction_6();
       private :
           int  Nombre1;
           int  Nombre2;
           char Lettre1;
           char Lettre2;
    };
```

Le second candidat pour devenir une classe d'objets est le regroupement Structure3_Globale_REGROUPER. Ce candidat possède les attributs : Nombre3, Nombre4, Lettre3 et Lettre4. De plus, ce candidat possède les méthodes : Main_B et Fonction_3_B.

```
class Structure3_Globale_REGROUPER
    {
```

```
public :
     int      Main_B();
     void     Fonction_3_B();
private :
     int  Nombre3;
     int  Nombre4;
     char Lettre3;
     char Lettre4;
};
```

Pour parvenir à effectuer ces regroupements, l'algorithme doit focaliser les fonctions `Main` et `Fonction_3`. Par conséquent, la fonction `Main` sera transformée en deux sous-fonctions, c'est-à-dire les sous-fonctions `Main_A` et `Main_B`. La sous-fonction `Main_A` pointe sur le nouveau type complexe de données `Structure1_Globale_REGROUPER`, tandis que la sous-fonction `Main_B` pointe sur le type complexe de données `Structure3_Globale_REGROUPER`. En ce qui concerne `Fonction_3`, c'est le même phénomème qui se produit. Cette fonction sera transformée en deux sous-fonctions, c'est-à-dire les sous-fonctions `Fonction_3_A` et `Fonction_3_B`. La sous-fonction `Fonction_3_A` pointe sur le type complexe de données `Structure1_Globale_REGROUPER`, tandis que la sous-fonction `Fonction_3_B` pointe sur le type complexe de données `Structure3_Globale_REGROUPER`. La figure 2.3 présente les regroupements produits par l'intervention d'un expert lors de l'application de l'algorithme d'identification des objets sur le graphe de visibilité des types du programme `Exemple1.cc`.

2.4.2 Le graphe de visibilité des données

Le graphe de références ne sera pas utile à l'identifcation des objets si le nombre de variables globales par rapport au nombre de fonctions n'est pas significatif. De plus, le graphe de références sera totalement inutile si on l'utilise pour l'identification des objets

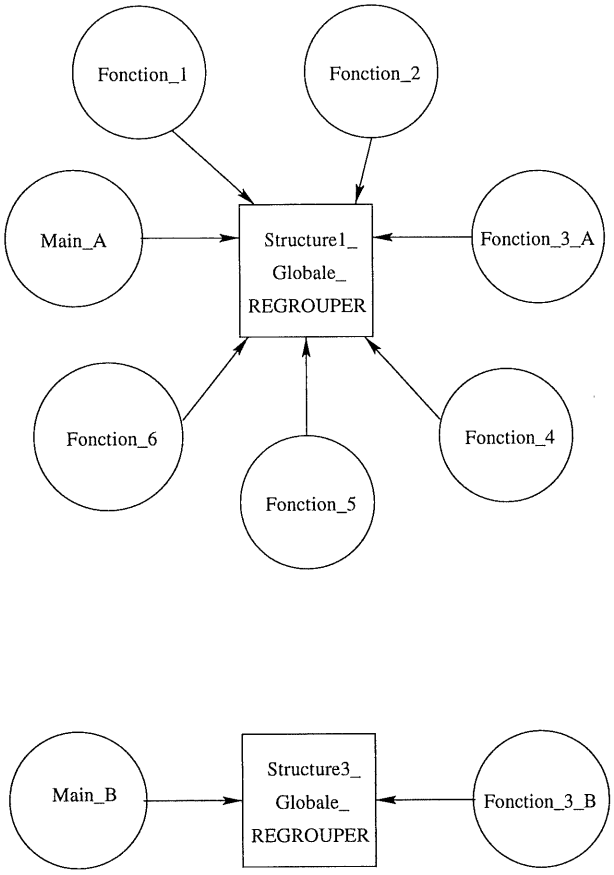

FIG. 2.3: Le graphe de visibilité des types avec l'intervention d'un expert lors de l'application de l'algorithme.

d'une librairie de fonctions. En effet, une librairie de fonctions ne possède pas de variables globales, car les données sont transmises entre les fonctions en utilisant des paramètres. Le graphe de visibilité des données est aussi un graphe bipartite. Un premier type de noeuds représente les données et un second type de noeuds représente les fonctions. Voici les deux ensembles à examiner pour créer les noeuds du graphe de visibilité des données :

- **DONNEES** : l'ensemble des données globales et locales à chaque fonction;
- **FONCTIONS** : l'ensemble des fonctions.

Les arcs indiquent la présence d'une donnée soit dans la déclaration, soit dans le corps de la fonction. Les noeuds et les arcs du graphe de visibilité des données sont décrits formellement par les deux ensembles ci-dessous :

- **NOEUDS** : $DONNEES \bigcup FONCTIONS$;
- **ARCS** : $\{(f, d) \mid f \in FONCTIONS \wedge d \in DONNEES \wedge f \ accède\ à\ d\}$.

Contrairement au graphe de références, qui considère uniquement les données déclarées globalement, le graphe de visibilité des données considère toutes les données globales du programme. Ces données globales sont celles déclarées globalement et celles déclarées localement mais passées par paramètre. La figure 2.4 présente le graphe de visibilité des données du programme `Exemple2.cc` (le code source de ce programme est disponible à l'annexe B). Il est important de noter que le nom de la variable a subi une légère modification nous permettant de savoir dans quelle fonction la variable a été déclarée locale. En effet, nous avons ajouté une extension au nom de la variable, s'il s'agit d'une variable locale à une fonction. Cette extension est composée du symbole " _ " suivi du nom de la fonction où elle a été déclarée localement.

L'utilisation du graphe de visibilité des données, en entrée à l'algorithme d'identification des objets présenté à la section 2.1.4, permet de produire des regroupements de fonctions et de données. Nous avons utilisé le graphe de visibilité des données du programme `Exemple2.cc`, en entrée à l'algorithme d'identification des objets. L'algorithme

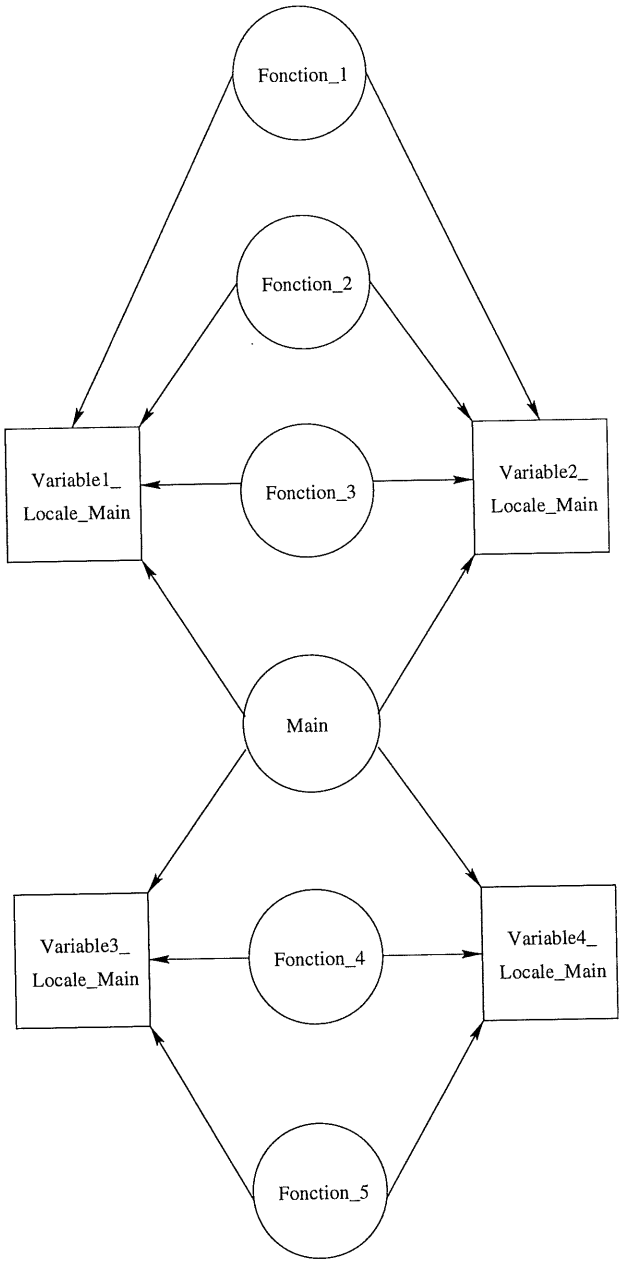

FIG. 2.4: Le graphe de visibilité des données du programme `Exemple2.cc`.

a permis de produire un regroupement composé de quatre variables. Le regroupement produit par l'algorithme est présenté à la figure 2.5.

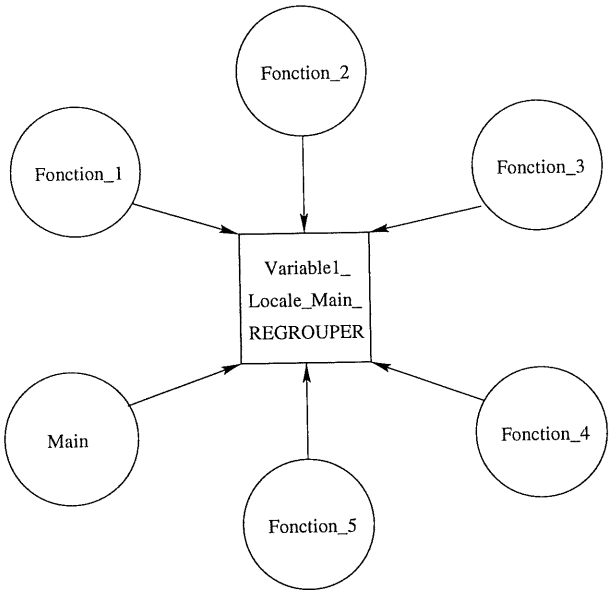

FIG. 2.5: Le graphe de visibilité des données après l'application de l'algorithme.

L'intervention de l'expert pour obtenir des meilleurs résultats

Par contre, un expert du programme `Exemple2.cc` pourrait identifier deux regroupements dans ce programme au lieu d'un seul. La figure 2.6 présente les regroupements qu'un expert suggère de faire sur le graphe de visibilité des données du programme `Exemple2.cc`.

L'intervention de l'expert lors de l'identification des objets dans le programme `Exemple2.cc` a permis de trouver deux candidats objets au lieu d'un seul. L'expert a modifié la proposition de l'algorithme dans le but d'obtenir des candidats objets plus significatifs. Bien entendu, le programme `Exemple2.cc` est un exemple simple et les résultats obtenus par l'algorithme sans l'intervention de l'expert ne sont pas nécessairement mauvais. La

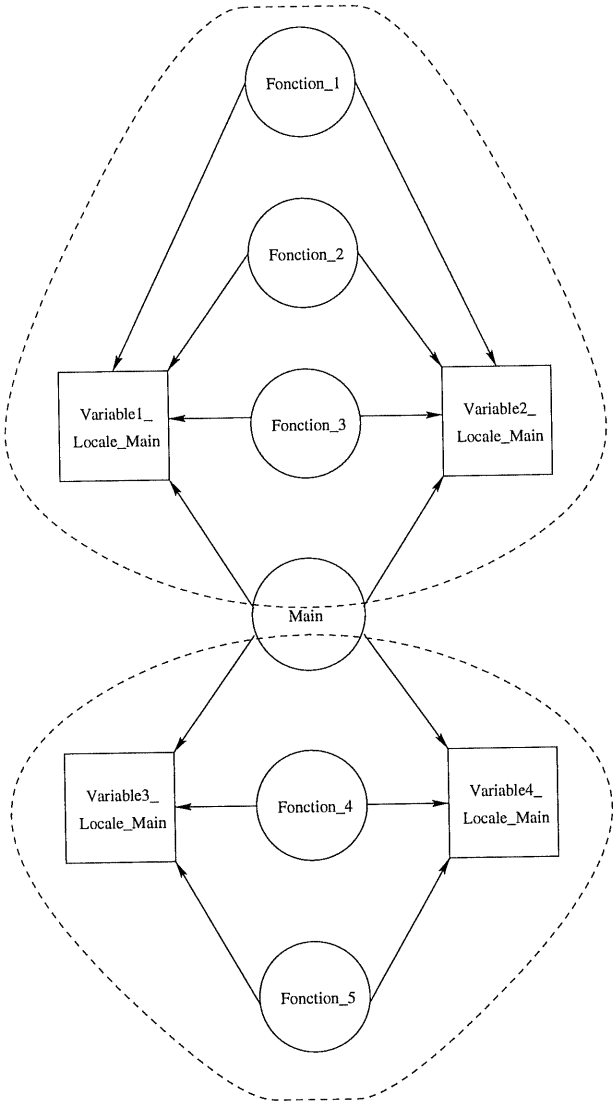

FIG. 2.6: Les regroupements possibles sur le graphe de visibilité des données.

figure 2.7 présente les deux candidats objets obtenus par l'intervention de l'expert lors de l'identification des objets dans le programme `Exemple2.cc`.

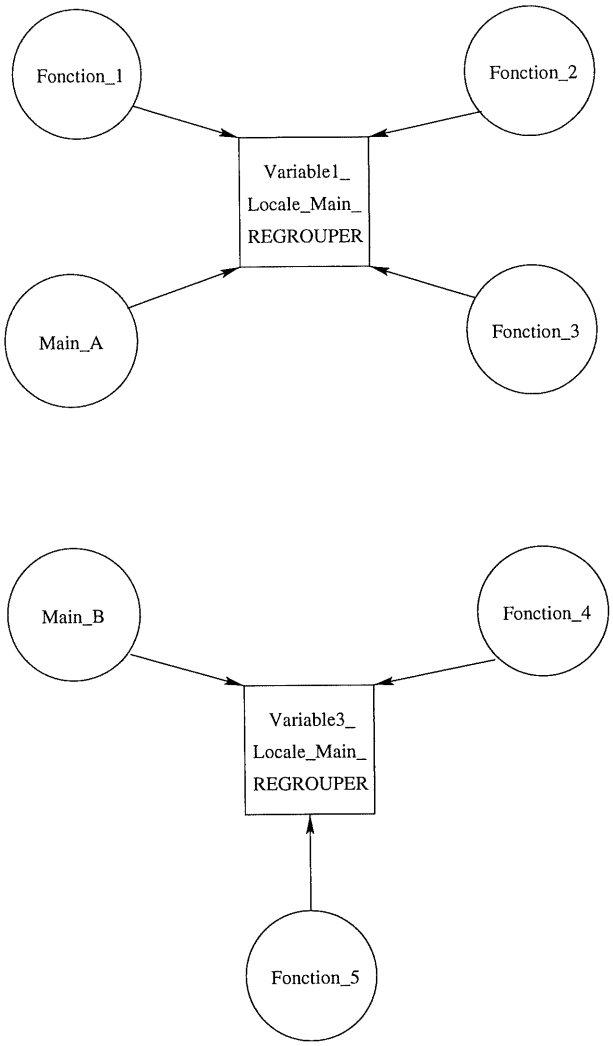

FIG. 2.7: Les candidats objets obtenus avec l'intervention d'un expert lors de l'application de l'algorithme.

2.4.3 Le graphe de références amélioré

Le graphe de références amélioré est un graphe de références identique à celui présenté à la section 2.1.1 mais avec la prise en compte du mode d'utilisation d'une variable glo-

bale. Cela permet de ne pas focaliser systématiquement une fonction qui touche à deux sous-graphes. Donc, chaque arc du graphe de références amélioré possède un paramètre. Ce paramètre est soit A pour indiquer que la variable est accédée par la fonction, soit M pour indiquer que la variable est modifiée par la fonction. Ce paramètre permet de prendre une décision concernant la focalisation d'une fonction qui touche à deux sous-graphes. La figure 2.8 présente un exemple d'une situation de focalisation. Il est possible d'observer que la fonction $FONCTION4$ appartient à deux sous-graphes. L'approche de Canfora et al. [4] suggère de focaliser systématiquement la fonction $FONCTION4$. Par contre, en considérant le mode d'utilisation de la variable par la fonction, il est possible de minimiser la focalisation. En effet, la figure 2.8 permet de conclure que la fonction $FONCTION4$ ne doit pas être focalisée. La fonction $FONCTION4$ doit appartenir au sous-graphe $SOUS\text{-}GRAPHE1$, car la fonction $FONCTION4$ ne modifie aucune variable appartenant au sous-graphe $SOUS\text{-}GRAPHE2$ et modifie une variable appartenant au sous-graphe $SOUS\text{-}GRAPHE1$, c'est-à-dire la variable $VARIABLE1$.

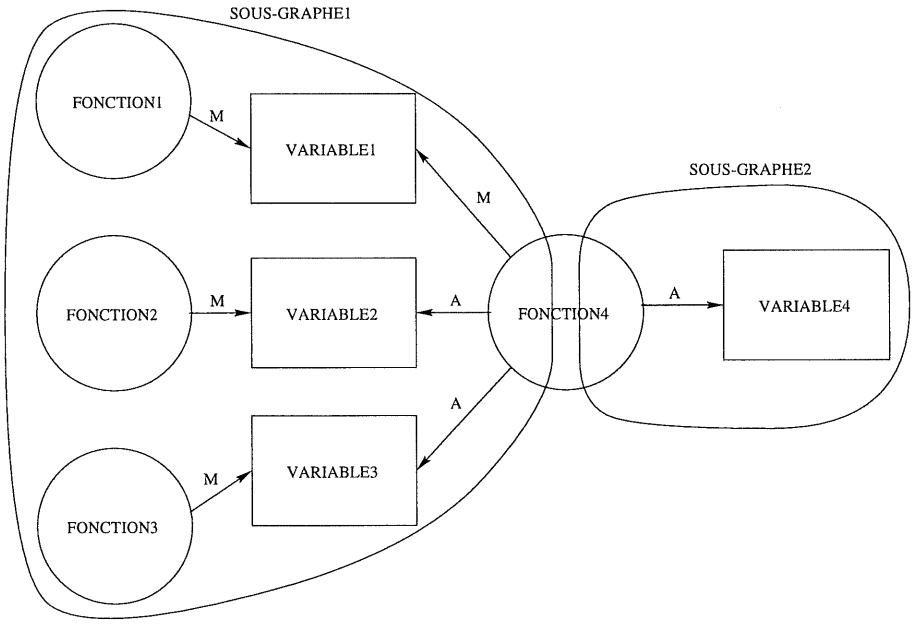

FIG. 2.8: Exemple d'une focalisation non systématique.

2.4.4 Notre manière de calculer le seuil

Le seuil est égal au $\Delta IC(f)$ maximum $-\epsilon$. Cette façon de calculer le seuil permet d'obtenir le plus grand nombre de regroupements possible, c'est-à-dire le plus grand nombre de candidats objets possible. De plus, elle permet d'obtenir des regroupements ayant le plus petit nombre d'attributs et de méthodes possible. Donc, cette façon de calculer le seuil permet d'obtenir des objets de petite taille. Cette manière de calculer le seuil a été établie à l'aide d'expérimentations. Les résultats obtenus en calculant le seuil de cette manière sont comparables aux résultats de Canfora et al. [4].

2.4.5 Une modification de la condition de fin de l'algorithme

Selon Canfora et al. [4], l'algorithme d'identification des objets se termine lorsque le graphe est transformé dans un ensemble de sous-graphes connexes, c'est-à-dire que chaque fonction du graphe a un ΔIC égal à zéro. Pour effectuer ces transformations sur le graphe, il faut regrouper des variables et focaliser des fonctions. La focalisation d'une fonction nécessite que celle-ci pointe sur une variable regroupée et un ensemble de variables non regroupées. Les expérimentations nous ont permis de constater que la condition normale de fin de l'algorithme ne peut pas toujours être respectée. En effet, il est possible que l'algorithme d'identification des objets se termine avec des fonctions possédant un ΔIC différent de zéro. Une fonction qui a un ΔIC égal à zéro est une fonction qui pointe sur une seule variable. Par ailleurs, nous avons rencontré un cas particulier, c'est-à-dire le cas où une fonction pointe sur deux variables ne devant pas être regroupées. Par conséquent, cette fonction ne peut pas être focalisée et l'algorithme se termine avec une fonction qui pointe sur deux variables. Il n'est pas justifié de regrouper ces deux variables. Donc, nous laissons le soin au concepteur de modifier la fonction manuellement.

Chapitre 3

LA RÉALISATION

L'identification des objets dans un code procédural est un processus qui doit être réalisé en deux phases. La première phase consiste à effectuer une rétro-ingénierie de l'application, c'est-à-dire à extraire les informations utiles à l'identification des objets dans le code procédural. La seconde phase consiste à exécuter l'algorithme d'identification des objets présenté à la section 2.1.4 en utilisant l'information extraite lors de la rétro-ingénierie. Ce chapitre présente les spécifications des outils permettant l'identification des objets, c'est-à-dire les extracteurs d'information du code source et le programme qui implante l'algorithme d'identification des objets. De plus, la section 3.3 montre un exemple de l'application de l'algorithme.

3.1 Les extracteurs de graphes

Cette section présente les spécifications des outils d'extraction du graphe de références, du graphe de références amélioré, du graphe de visibilité des types et du graphe de visibilité des données. Il est important de noter que les extracteurs sont conçus pour extraire de l'information à partir d'un code source écrit en langage C. Les extracteurs sont construits sous forme de règles applicables sur l'arbre syntaxique du programme source. Ces règles permettent d'extraire des faits du programme source. Les faits peuvent être

combinés pour former la base de faits utilisée par l'algorithme d'identification des objets. Pour bien saisir le principe de fonctionnement des outils d'extraction, nous présenterons les techniques de compilation utiles à l'analyse d'un code source.

3.1.1 L'analyse du code source

Les principes et les techniques du développement de compilateurs sont importants lors de la réingénierie d'une application. En effet, la phase de rétro-ingénierie de l'application exige l'extraction des informations du code source. L'outil d'extraction est implanté selon les principes et les techniques de développement des compilateurs. Il doit créer l'arbre syntaxique du programme pour lequel il veut extraire de l'information. La création de l'arbre syntaxique est réalisée à l'aide de l'analyse lexicale et de l'analyse syntaxique. Lorsque l'arbre syntaxique est créé, l'outil d'extraction applique des patrons sur cet arbre pour identifier l'information qu'il doit extraire. Bien entendu, l'utilisation d'une grammaire hors contexte est requise pour le codage des patrons, car l'arbre syntaxique est construit selon cette grammaire. Finalement, l'outil d'extraction produit un fichier représentant une base de faits. Ces faits sont utilisés en entrée à l'algorithme d'identification des objets présenté à la section 2.1.4.

3.1.2 Les règles générales aux trois extracteurs de graphes

Les règles sont composées d'un antécédent et d'un conséquent. D'abord, il faut évaluer si l'antécédent est vrai ou faux. Lorsque l'antécédent est vrai le conséquent est exécuté. L'antécédent est une condition sur la base de faits et la base de faits est dynamique. Donc, si le conséquent crée un nouveau fait, alors ce fait est considéré par l'antécédent de chaque règle.

Règle 1 : Extraire le type et le nom des variables globales

Extraire les variables globales déclarées avec des types simples

Extraire les variables globales déclarées avec des types complexes

Extraire les variables globales déclarées sous forme de listes

POUR-CHAQUE variable globale

FAIRE

Créer le fait VARIABLE_GLOBALE(TYPE, NOM)

FIN-POUR-CHAQUE

Règle 2 : Extraire le nom et le corps des fonctions

POUR-CHAQUE fonction

FAIRE

Créer le fait FONCTION(NOM, CORPS)

Créer le fait DECLARATION_FONCTION(NOM, PARAMETRES)

FIN-POUR-CHAQUE

Règle 3 : Extraire le nom des types complexes

Extraire les types complexes construits à l'aide d'un STRUCT

Extraire les types complexes construits à l'aide d'un UNION

Extraire les types complexes construits à l'aide d'un TYPEDEF

POUR-CHAQUE type complexe

FAIRE

Créer le fait TYPE_COMPLEXE(NOM)

FIN-POUR-CHAQUE

3.1.3 L'extracteur du graphe de références

Règle 4 : Extraire le graphe de références

SI

FONCTION(IDENTIFIANT1, CORPS) **ET**

VARIABLE_GLOBALE(TYPE, IDENTIFIANT2) **ET**

IDENTIFIANT2 se retrouve dans CORPS

ALORS

Créer le fait REFERENCE(IDENTIFIANT1, IDENTIFIANT2, TYPE)

FIN-SI

3.1.4 L'extracteur du graphe de références amélioré

Règle 5 : Extraire le graphe de références amélioré

SI

REFERENCE(IDENTIFIANT1, IDENTIFIANT2, TYPE) **ET**

IDENTIFIANT2 se retrouve à gauche d'un opérateur dans le corps de IDENTIFIANT1

(il y a d'autres façons de modifier une variable qui ne sont pas traitées)

ALORS

Créer le fait AMÉLIORER(IDENTIFIANT1, IDENTIFIANT2, TYPE, M)

SINON

Créer le fait AMÉLIORER(IDENTIFIANT1, IDENTIFIANT2, TYPE, A)

FIN-SI

3.1.5 L'extracteur du graphe de visibilité des types

Règle 6 : Extraire le graphe de visibilité des types lorsque le type provient d'une variable globale

SI

FONCTION(IDENTIFIANT1, CORPS) **ET**

VARIABLE_GLOBALE(TYPE, IDENTIFIANT2) **ET**

TYPE_COMPLEXE(TYPE) **ET**

IDENTIFIANT2 se retrouve dans CORPS

ALORS

Créer le fait VISIBILITE_TYPES(IDENTIFIANT1, TYPE, "GLOBALE")

FIN-SI

Règle 7 : Extraire le graphe de visibilité des types lorsque le type provient d'une variable locale

SI

FONCTION(IDENTIFIANT1, CORPS) **ET**

TYPE_COMPLEXE(TYPE) **ET**

TYPE se retrouve dans CORPS

ALORS

Créer le fait VISIBILITE_TYPES(IDENTIFIANT1, TYPE, "LOCALE")

FIN-SI

Règle 8 : Extraire le graphe de visibilité des types lorsque le type provient d'un paramètre formel

SI

DECLARATION_FONCTION(IDENTIFIANT1, PARAMETRES) **ET**

TYPE_COMPLEXE(TYPE) **ET**

TYPE se retrouve dans PARAMETRES

ALORS

Créer le fait VISIBILITE_TYPES(IDENTIFIANT1, TYPE, "PARAMETRE")

FIN-SI

3.1.6 L'extracteur du graphe de visibilité des données

Règle 9 : Extraire les variables locales à chaque fonction

Extraire les variables locales déclarées avec des types simples

Extraire les variables locales déclarées avec des types complexes

POUR-CHAQUE variable locale de chaque fonction

FAIRE

Créer le fait VARIABLE_LOCALE(FONCTION, VARIABLE, TYPE)

FIN-POUR-CHAQUE

Règle 10 : Extraire les appels de fonctions

POUR-CHAQUE appel de fonction

FAIRE

Créer le fait APPEL_FONCTION(FONCTION1, FONCTION2)

La FONCTION1 est la fonction appelante

La FONCTION2 est la fonction appelée

FIN-POUR-CHAQUE

Règle 11 : Extraire les paramètres formels

POUR-CHAQUE paramètre formel

FAIRE

Créer le fait PARAMETRE_FORMEL(FONCTION, VARIABLE, RANG)

Le RANG est la position du paramètre lors de la déclaration de la fonction

FIN-POUR-CHAQUE

Règle 12 : Extraire les paramètres réels

POUR-CHAQUE paramètre réel

FAIRE

Créer le fait PARAMETRE_REEL(FONCTION1, FONCTION2, VARIABLE, RANG)

La FONCTION1 est la fonction appelante

La FONCTION2 est la fonction appelée

Le RANG est la position du paramètre lors de l'appel de la fonction

FIN-POUR-CHAQUE

Règle 13 : Extraire le graphe de visibilité des données — PARTIE 1

SI

PARAMETRE_REEL(FONCTION1, FONCTION2, VARIABLE1, RANG1) **ET**

VARIABLE_LOCALE(FONCTION1, VARIABLE1, TYPE) **ET**

PARAMETRE_FORMEL(FONCTION2, VARIABLE2, RANG1)

ALORS

VARIABLE3 = VARIABLE1 + "_" + FONCTION1

Créer le fait VISIBILITE_DONNEES(FONCTION1, VARIABLE3, TYPE)

Créer le fait VISIBILITE_DONNEES(FONCTION2, VARIABLE3, TYPE)

Créer le fait LIEN(VARIABLE3, FONCTION1, VARIABLE2, FONCTION2, TYPE, RANG1)

FIN-SI

Règle 14 : Extraire le graphe de visibilité des données — PARTIE 2

SI

LIEN(VARIABLE1, FONCTION1, VARIABLE2, FONCTION2, TYPE, RANG1) **ET**

PARAMETRE_REEL(FONCTION2, FONCTION3, VARIABLE2, RANG2) **ET**

PARAMETRE_FORMEL(FONCTION3, VARIABLE3, RANG2)

ALORS

Créer le fait VISIBILITE_DONNEES(FONCTION3, VARIABLE1, TYPE)

Créer le fait LIEN(VARIABLE1, FONCTION2, VARIABLE3, FONCTION3, TYPE, RANG2)

FIN-SI

3.2 L'algorithme d'identification des objets dans un code source procédural

Cette section présente les spécifications du programme qui implante l'algorithme d'identification des objets présenté à la section 2.1.4.

3.2.1 La procédure principale

Extraire l'information de la base de faits

Créer le graphe à partir des informations extraites

FAIRE

Afficher le graphe

POUR-CHAQUE noeud fonction F du graphe

FAIRE

Calculer le $\Delta IC(F)$

FIN-POUR-CHAQUE

Calculer le seuil

Calculer l'ensemble REGROUPER

Calculer l'ensemble FOCALISER

Afficher les ensembles REGROUPER et FOCALISER

SI l'expert désire modifier le graphe

ALORS

Effectuer le regroupement

Effectuer la focalisation

FIN-SI

TANT-QUE $\Delta IC(F)$ de chaque fonction <> 0

3.2.2 Extraire l'information de la base de faits

Ouvrir le fichier $F1$ représentant la base de faits en lecture

TANT-QUE la fin du fichier $F1$ n'est pas atteinte

FAIRE

POUR-CHAQUE ligne du fichier $F1$

FAIRE

Extraire le nom de la fonction dans $TAB\_FICHIER[no\_ligne].FONC-TION$

Extraire le nom de la donnée $TAB\_FICHIER[no\_ligne].DONNEE$

Extraire le type de la donnée $TAB\_FICHIER[no\_ligne].TYPE$

FIN-POUR-CHAQUE

FIN-TANT-QUE

3.2.3 Créer le graphe à partir des informations extraites

POUR-CHAQUE indice de $TAB\_FICHIER$

FAIRE

SI le noeud fonction F n'existe pas

ALORS

Créer le noeud fonction F

FIN-SI

SI le noeud de donnée D n'existe pas

ALORS

Créer le noeud de donnée D

FIN-SI

SI l'arc entre le noeud de fonction F et le noeud de donnée D n'existe pas

ALORS

Créer l'arc entre le noeud de fonction F et le noeud de donnée D

FIN-SI

FIN-POUR-CHAQUE

Créer le noeud de fonction F

Alimenter le tableau de noeuds fonctions avec le nom de la fonction

Créer le noeud de donnée D

Alimenter le tableau de noeuds données avec le nom et le type de la donnée

Créer l'arc entre le noeud de fonction F et le noeud de donnée D

Ajouter le noeud de donnée D dans la liste des données pointées par la fonction F

Ajouter le noeud de fonction F dans la liste des fonctions pointées par la donnée D

3.2.4 Afficher le graphe

POUR-CHAQUE noeud de fonction F

FAIRE

Afficher le nom de la fonction F

Afficher le nom et le type de chaque donnée D pointée par la fonction F

FIN-POUR-CHAQUE

3.2.5 Calculer le $\Delta IC(F)$

Calculer le $IC(F)$

Calculer la partie de droite de l'expression arithmétique

$\Delta IC(F) = IC(F) - Partie\_de\_droite$

Calculer le $IC(F)$

Calculer le NUMÉRATEUR de l'expression arithmétique

POUR-CHAQUE noeud de donnée D pointé par F

FAIRE

POUR-CHAQUE noeud de fonction F_i pointé par D

FAIRE

SI l'ensemble des données pointées par F_i est inclus dans l'ensemble des données pointées par F

ALORS

Incrémenter le compteur NUMÉRATEUR

FIN-SI

FIN-POUR-CHAQUE

Calculer le DÉNOMINATEUR de l'expression arithmétique

POUR-CHAQUE noeud de donnée D pointé par F

FAIRE

DÉNOMINATEUR = au nombre de noeuds fonctions F_i pointé par D

FIN-POUR-CHAQUE

$IC(F)$ = NUMÉRATEUR $\div$ DÉNOMINATEUR

Calculer la partie de droite de l'expression arithmétique

POUR-CHAQUE noeud de donnée D pointé par F

FAIRE

Calculer le NUMÉRATEUR de l'expression arithmétique

POUR-CHAQUE noeud de fonction F_i pointé par D

FAIRE

SI F_i pointée par D pointe uniquement sur D

ALORS

Incrémenter le compteur NUMÉRATEUR

FIN-SI

FIN-POUR-CHAQUE

Calculer le DÉNOMINATEUR de l'expression arithmétique

DÉNOMINATEUR = au nombre de noeuds fonctions F_i pointé par D

Partie_de_droite = NUMÉRATEUR $\div$ DÉNOMINATEUR

FIN-POUR-CHAQUE

3.2.6 Calculer le seuil

Seuil = $\Delta IC(F)$ maximum $-$ 0.001

3.2.7 Calculer l'ensemble REGROUPER

POUR-CHAQUE noeud de fonction F

FAIRE

SI $\Delta IC(F) > Seuil$

ALORS

Ajouter le noeud de fonction F dans la liste des fonctions à regrouper

FIN-SI

FIN-POUR-CHAQUE

3.2.8 Calculer l'ensemble FOCALISER

POUR-CHAQUE noeud de fonction F

FAIRE

SI $\Delta IC(F) <= Seuil \wedge \Delta IC(F) \neq 0$

ALORS

Ajouter le noeud de fonction F dans la liste des fonctions à focaliser

FIN-SI

FIN-POUR-CHAQUE

3.2.9 Afficher les ensembles REGROUPER et FOCALISER

Afficher la liste des fonctions F à regrouper

Afficher la liste des fonctions F à focaliser

3.2.10 Effectuer le regroupement

POUR-CHAQUE noeud de fonction F appartenant à la liste des fonctions à regrouper

FAIRE

Ajouter les données D pointées par F dans la liste $L1$ des données à regrouper

FIN-POUR-CHAQUE

Créer le nouveau noeud de donnée D_{L1} en regroupant les données de la liste $L1$

Modifier les arcs des noeuds fonctions F appartenant à la liste des fonctions à regrouper pour qu'elles pointent sur le nouveau noeud de donnée D_{L1}

Ajouter les arcs au nouveau noeud de donnée D_{L1} pour qu'il pointe sur les noeuds fonctions F appartenant à la liste des fonctions à regrouper

3.2.11 Effectuer la focalisation

POUR-CHAQUE noeud de fonction F appartenant à la liste des fonctions à focaliser

FAIRE

SI la fonction F est vraiment à focaliser, c'est-à-dire que F doit pointer sur un nouveau noeud de donnée D_{L1} et sur un ou plusieurs noeuds données D

ALORS

Ajouter la fonction F dans la liste $L2$ des fonctions vraiment à focaliser

FIN-SI

FIN-POUR-CHAQUE

POUR-CHAQUE noeud de fonction F appartenant à la liste $L2$ des fonctions vraiment à focaliser

FAIRE

SI le type de graphe utilisé en entrée est le graphe de références amélioré

ALORS

SI le noeud de donnée D_{L2} est modifié par la fonction F et au moins une des données D, autres que D_{L2}, est modifiée par la fonction F

ALORS

Créer le noeud de fonction F_A

Ajouter l'arc entre la fonction F_A et le nouveau noeud de donnée D_{L2}

Créer le noeud de fonction F_B

Ajouter les arcs entre la fonction F_B et les données D pointées par F

Détruire le noeud de fonction F

SINON

SI le noeud de donnée D_{L2} est accédé par la fonction F et au moins une des données D est modifiée par la fonction F

ALORS

Faire pointer la fonction F uniquement sur les données D

SINON

Faire pointer la fonction F uniquement sur le noeud de donnée D_{L2}, car il s'agit du cas où F ne modifie aucune donnée

FIN-SI

FIN-SI

SINON

Il s'agit du cas où le type de graphe utilisé en entrée n'est pas le graphe de références amélioré

Créer le noeud de fonction F_A

Ajouter l'arc entre la fonction F_A et le nouveau noeud de donnée D_{L2}

Créer le noeud de fonction F_B

Ajouter les arcs entre la fonction F_B et les données D pointées par F

Détruire le noeud de fonction F

FIN-SI

FIN-POUR-CHAQUE

3.3 L'exemple

Cette section présente un exemple de l'application de l'algorithme d'identification des objets dans un code source procédural. L'exemple utilisé est connu, de petite taille et se nomme `Exemple3.cc`. Il s'agit d'un système qui implante la gestion d'une pile, d'une file et d'une liste (le code source de l'application est disponible à l'annexe C). L'identification des objets est réalisée en deux étapes : la création de la base de faits et l'exécution de l'algorithme d'identification des objets. Cet exemple est présenté uniquement pour le graphe de références du programme `Exemple3.cc`.

3.3.1 La base de faits

La première étape consiste à utiliser un outil d'analyse syntaxique pour extraire les faits permettant de construire le graphe de références du système. Notre outil se nomme SEM et nous est fourni par notre partenaire industriel. Un fait est formé de trois paramètres, à savoir le nom de la fonction, le nom de la variable et le type de la variable. Chaque fait représente un arc entre un noeud de type fonction et un noeud de type variable. Voici la base de faits extraite, par notre outil d'analyse syntaxique, du programme `Exemple3.cc`.

```
REFERENCE("stack_push", "stack_struct", "ELEM_T")
REFERENCE("stack_push", "stack_point", "int")
REFERENCE("stack_pop", "stack_struct", "ELEM_T")
REFERENCE("stack_pop", "stack_point", "int")
REFERENCE("stack_top", "stack_struct", "ELEM_T")
REFERENCE("stack_top", "stack_point", "int")
REFERENCE("stack_Empty", "stack_point", "int")
REFERENCE("stack_full", "stack_point", "int")
REFERENCE("queue_insert", "queue_struct", "ELEM_T")
REFERENCE("queue_insert", "queue_head", "int")
```

```
REFERENCE("queue_insert", "queue_num_elem", "int")
REFERENCE("queue_extract", "queue_struct", "ELEM_T")
REFERENCE("queue_extract", "queue_tail", "int")
REFERENCE("queue_extract", "queue_num_elem", "int")
REFERENCE("queue_Empty", "queue_num_elem", "int")
REFERENCE("queue_full", "queue_num_elem", "int")
REFERENCE("list_add", "list", "list_struct")
REFERENCE("list_elim", "list", "list_struct")
REFERENCE("list_is_in", "list", "list_struct")
REFERENCE("list_empty", "list", "list_struct")
REFERENCE("global_init", "stack_point", "int")
REFERENCE("global_init", "queue_head", "int")
REFERENCE("global_init", "queue_tail", "int")
REFERENCE("global_init", "queue_num_elem", "int")
REFERENCE("global_init", "list", "list_struct")
REFERENCE("stack_to_list", "stack_struct", "ELEM_T")
REFERENCE("stack_to_list", "stack_point", "int")
REFERENCE("stack_to_list", "list", "list_struct")
REFERENCE("stack_to_queue", "stack_struct", "ELEM_T")
REFERENCE("stack_to_queue", "queue_struct", "ELEM_T")
REFERENCE("stack_to_queue", "stack_point", "int")
REFERENCE("stack_to_queue", "queue_head", "int")
REFERENCE("stack_to_queue", "queue_num_elem", "int")
REFERENCE("queue_to_stack", "stack_struct", "ELEM_T")
REFERENCE("queue_to_stack", "queue_struct", "ELEM_T")
REFERENCE("queue_to_stack", "stack_point", "int")
REFERENCE("queue_to_stack", "queue_tail", "int")
REFERENCE("queue_to_stack", "queue_num_elem", "int")
```

```
REFERENCE("queue_to_list", "queue_struct", "ELEM_T")
REFERENCE("queue_to_list", "queue_tail", "int")
REFERENCE("queue_to_list", "queue_num_elem", "int")
REFERENCE("queue_to_list", "list", "list_struct")
REFERENCE("list_to_stack", "stack_struct", "ELEM_T")
REFERENCE("list_to_stack", "stack_point", "int")
REFERENCE("list_to_stack", "list", "list_struct")
REFERENCE("list_to_queue", "queue_struct", "ELEM_T")
REFERENCE("list_to_queue", "queue_head", "int")
REFERENCE("list_to_queue", "queue_num_elem", "int")
REFERENCE("list_to_queue", "list", "list_struct")
```

3.3.2 L'algorithme d'identification des objets

La deuxième étape consiste à exécuter le programme, implantant l'algorithme d'identification des objets, en utilisant comme entrée la base de faits extraite précédemment. L'exécution du programme requiert deux itérations de l'algorithme pour parvenir à identifier les candidats objets. La figure 3.1 présente le graphe de références du programme `Exemple3.cc` avant de lancer l'algorithme d'identification des objets.

Première itération

Pour décomposer le graphe de références en sous-graphes connexes, l'outil d'identification des objets doit évaluer la variation de l'indice de connectivité (ΔIC) de chaque fonction. Voici les valeurs de cet indice pour chacune des fonctions du système.

$\Delta IC(stack\_push) = 0,270588$

$\Delta IC(stack\_pop) = 0,270588$

$\Delta IC(stack\_top) = 0,270588$

$\Delta IC(stack\_empty) = 0$

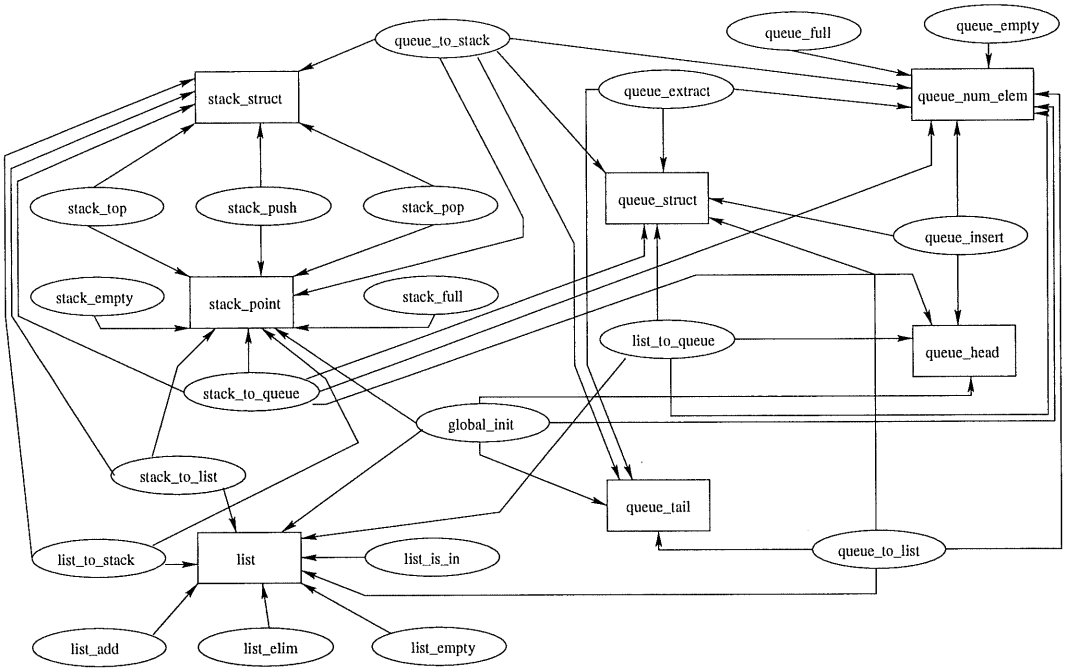

FIG. 3.1: Graphe de références du programme `Exemple3.cc`.

$\Delta IC(stack\_full) = 0$

$\Delta IC(queue\_insert) = 0,0409357$

$\Delta IC(queue\_extract) = 0,0409357$

$\Delta IC(queue\_empty) = 0$

$\Delta IC(queue\_full) = 0$

$\Delta IC(list\_add) = 0$

$\Delta IC(list\_elim) = 0$

$\Delta IC(list\_is\_in) = 0$

$\Delta IC(list\_empty) = 0$

$\Delta IC(stack\_to\_list) = 0,0478632$

$\Delta IC(stack\_to\_queue) = 0,0777778$

$\Delta IC(queue\_to\_stack) = 0,0777778$

$\Delta IC(queue\_to\_list) = -0,202381$

$\Delta IC(list\_to\_stack) = 0,0478632$

$\Delta IC(list\_to\_queue) = -0,202381$

$\Delta IC(global\_init) = -0,505556$

Après avoir calculer le ΔIC de chaque fonction, l'outil détermine le seuil (ΔIC maximum $-$ 0.001). Le seuil pour la première itération est de $0,269588$. Le calcul du seuil permet d'identifier l'ensemble des fonctions devant être regroupées et l'ensemble des fonctions devant être focalisées. Voici les deux ensembles produits par l'outil :

Fonctions à regrouper

1. *stack_push*
2. *stack_pop*
3. *stack_top*

Fonctions à focaliser

1. *queue_insert*
2. *queue_extract*
3. *stack_to_list*
4. *stack_to_queue*
5. *queue_to_stack*
6. *queue_to_list*
7. *list_to_stack*
8. *list_to_queue*
9. *global_init*

Le regroupement des fonctions

stack_push, *stack_pop* et *stack_top*

permet de réunir les variables

stack_struct et *stack_point*

pour former la nouvelle variable *stack*. La nouvelle variable *stack* est de type complexe, car elle est formée de deux variables de type simple. De plus, le regroupement des fonctions

stack_push, *stack_pop* et *stack_top*

permet de déterminer les fonctions devant être réellement focalisées. Une fonction est dite à focaliser réellement si après le regroupement, elle pointe sur une nouvelle variable regroupée et d'autres variables. Voici l'ensemble des fonctions devant être réellement focalisées :

Fonctions à focaliser réellement

1. *stack_to_list*
2. *stack_to_queue*
3. *queue_to_stack*
4. *list_to_stack*
5. *global_init*

La figure 3.2 présente le graphe de références du programme `Exemple3.cc` après la première itération.

Deuxième itération

A chaque itération, l'outil calcule à nouveau la valeur du ΔIC de chaque fonction et détermine le nouveau seuil. Le seuil pour la deuxième itération est de $0,1978304$. Voici l'ensemble des fonctions à regrouper et l'ensemble des fonctions à focaliser produits par l'outil :

Fonctions à regrouper

1. *queue_insert*

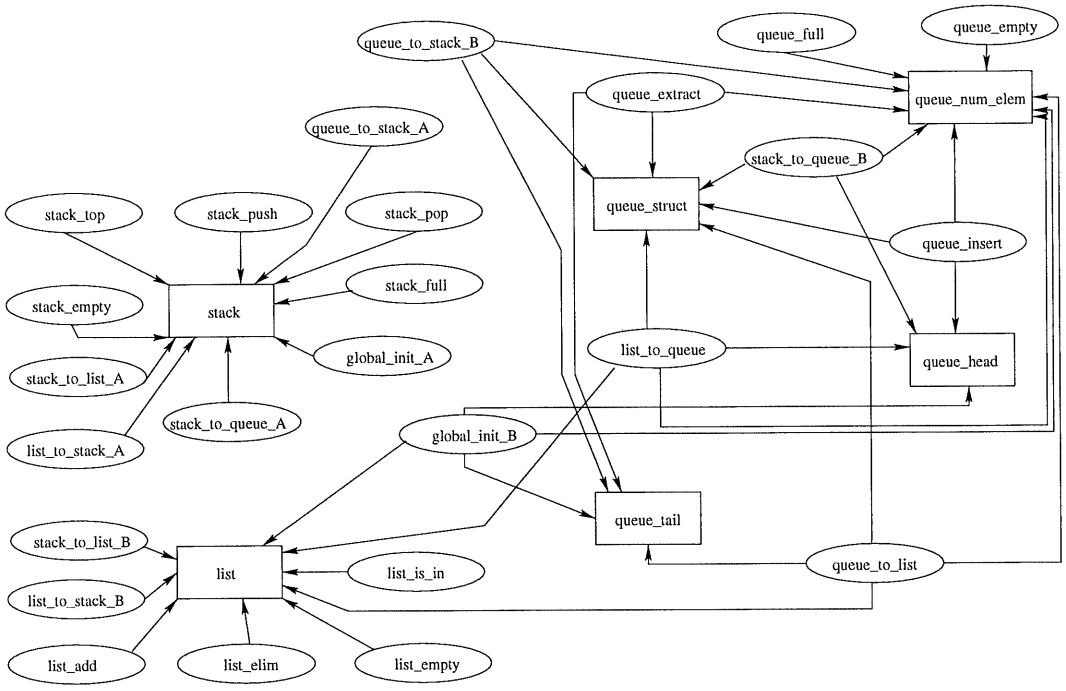

FIG. 3.2: Graphe de références après la première itération.

2. *queue_extract*
3. *stack_to_queue_B*
4. *queue_to_stack_B*

Fonctions à focaliser

1. *queue_to_list*
2. *list_to_queue*
3. *global_init_B*

Le regroupement des fonctions

queue_insert, *queue_extract*, *stack_to_queue_B* et *queue_to_stack_B*

permet de réunir les variables

queue_struct, *queue_num_elem*, *queue_head* et *queue_tail*

pour former la nouvelle variable *queue*. De plus, le regroupement des fonctions

queue_insert, *queue_extract*, *stack_to_queue_B* et *queue_to_stack_B*

permet de déterminer les fonctions devant être focalisées réellement. Voici l'ensemble des fonctions devant être focalisées réellement :

Fonctions à focaliser réellement

1. *queue_to_list*
2. *list_to_queue*
3. *global_init_B*

La figure 3.3 présente le graphe de références du programme `Exemple3.cc` après la deuxième itération. Ce graphe permet de visualiser les trois sous-graphes produits par l'algorithme d'identification des objets. Ensuite, la figure 3.4 présente les candidats objets identifiés dans le programme `Exemple3.cc` à l'aide de l'algorithme de décomposition de graphes.

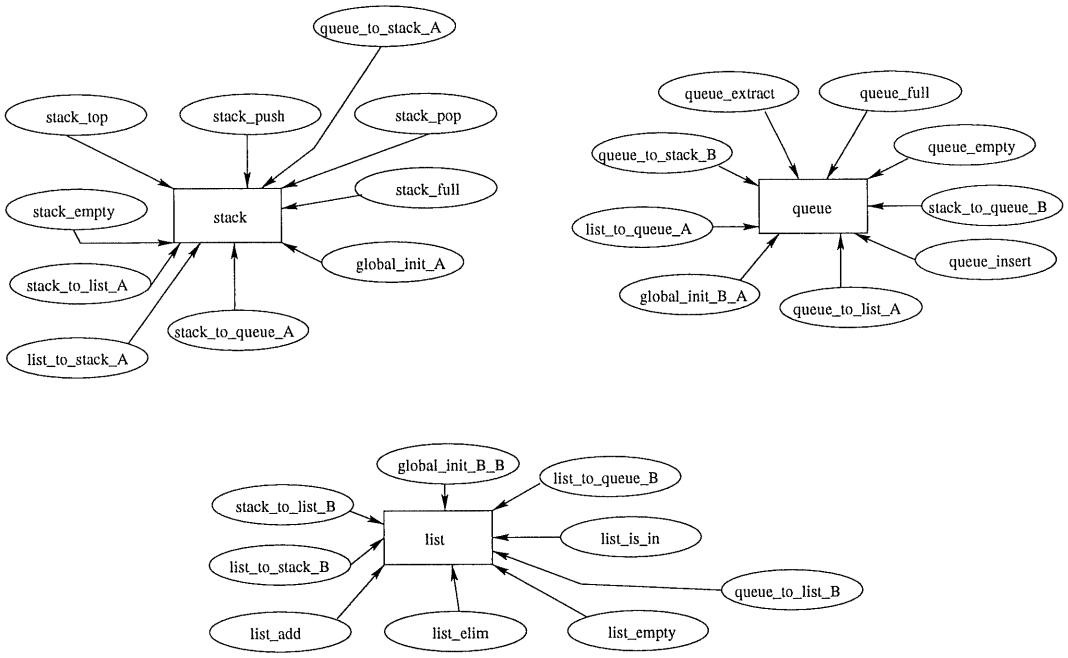

FIG. 3.3: Graphe de références après la deuxième itération.

STACK
ELEM_T Stack_struct INT Stack_point
Stack_top Stack_push Stack_pop Stack_empty Stack_full Stack_to_list_A List_to_stack_A Stack_to_queue_A Queue_to_stack_A Global_init_A

QUEUE
ELEM_T Queue_struct INT Queue_num_elem INT Queue_head INT Queue_tail
Queue_insert Queue_extract Queue_full Queue_empty Queue_to_stack_B Stack_to_queue_B Queue_to_list_A List_to_queue_A Global_init_B_A

LIST
LIST_STRUCT List
List_add List_elim List_is_in List_empty List_to_stack_B Stack_to_list_B List_to_queue_B Queue_to_list_B Global_init_B_B

FIG. 3.4: Les objets identifiés dans le programme `Exemple3.cc`.

Chapitre 4

L'ÉTUDE DE CAS

Ce chapitre présente les résultats obtenus lors de l'identification des objets dans trois systèmes informatiques de grande envergure. Il s'agit de trois systèmes développés au Centre de recherche informatique de Montréal par l'unité Systèmes à Base de Connaissances (SBC). Nous avons choisi d'utiliser ces trois systèmes pour valider notre approche, car les gens qui ont implanté ces systèmes étaient disponibles pour valider nos résultats. Les sections 4.1, 4.2 et 4.3 montrent l'ampleur des systèmes, ainsi que les résultats obtenus. Finalement, la section 4.4 présente la validation des résultats par les experts humains.

4.1 Le système SBC1

Cette section présente les résultats obtenus lors de l'identification des objets dans le système `SBC1`. Le tableau 4.1 présente la taille du système `SBC1` en considérant le nombre de noeuds de type donnée, le nombre de noeuds de type fonction, le nombre de méthodes et le nombre d'objets identifiés avec chacun des graphes utilisés.

SBC1

Nombre de lignes de code source = 43 308				
Graphe	**Noeuds données**	**Noeuds fonctions**	**Méthodes identifiées**	**Objets identifiés**
Graphe de références	14	18	18	10
Graphe de références amélioré	14	18	18	10
Graphe de visibilité des types	11	219	254	7
Graphe de visibilité des données	68	57	97	22

TAB. 4.1: Objets identifiés dans le système SBC1.

4.2 Le système SBC2

Cette section présente les résultats obtenus lors de l'identification des objets dans le système `SBC2`. Le tableau 4.2 présente la taille du système `SBC2` en considérant le nombre de noeuds de type donnée, le nombre de noeuds de type fonction, le nombre de méthodes et le nombre d'objets identifiés avec chacun des graphes utilisés.

SBC2

Nombre de lignes de code source = 5 851				
Graphe	**Noeuds données**	**Noeuds fonctions**	**Méthodes identifiées**	**Objets identifiés**
Graphe de références	26	28	40	10
Graphe de références amélioré	26	28	32	11
Graphe de visibilité des types	19	90	126	16
Graphe de visibilité des données	88	65	114	24

TAB. 4.2: Objets identifiés dans le système SBC2.

4.3 Le système SBC3

Cette section présente les résultats obtenus lors de l'identification des objets dans le système SBC3. Le tableau 4.3 présente la taille du système SBC3 en considérant le nombre de noeuds de type donnée, le nombre de noeuds de type fonction, le nombre de méthodes et le nombre d'objets identifiés avec chacun des graphes utilisés.

SBC3

Nombre de lignes de code source = 25 123				
Graphe	**Noeuds données**	**Noeuds fonctions**	**Méthodes identifiées**	**Objets identifiés**
Graphe de références	23	61	66	17
Graphe de références amélioré	23	61	61	17
Graphe de visibilité des types	51	484	726	36
Graphe de visibilité des données	206	187	284	61

TAB. 4.3: Objets identifiés dans le système SBC3.

4.4 La validation des résultats

Les sous-sections suivantes présentent la validation des résultats obtenus lors de l'identification des objets dans les systèmes SBC1, SBC2 et SBC3. Ces validations ont été réalisées par les personnes qui ont implanté les systèmes. Notre façon de procéder consiste à identifier les objets dans le code source d'une application et de valider nos résultats avec un expert humain. Le rôle de cet expert est de déterminer dans quelle proportion les objets que nous avons identifiés sont réellement des objets. Par conséquent, il est clair que les objets que notre approche n'a pas identifiés ne sont pas considérés par l'expert. Donc, une approche de validation qui permettrait de construire un modèle d'analyse orienté ob-

jets de l'application pourrait être avantageuse. En effet, ce type d'approche permettrait d'évaluer non seulement l'exactitude des objets que nous avons trouvés mais d'évaluer le pourcentage des objets que nous avons trouvés.

4.4.1 La validation de SBC1

Le tableau 4.4 présente la validation des résultats obtenus lors de l'identification des objets dans le système `SBC1`. On constate que le pourcentage d'objets identifiés correctement avec le graphe de références et le graphe de références amélioré est 80 %. Bien que ces ratios soient significatifs, il ne faut pas oublier qu'il s'agit uniquement des objets que notre approche est parvenue à identifier. En effet, une analyse orientée objets du système `SBC1` aurait pu permettre d'identifier plus de dix objets pour un tel système.

Les résultats obtenus avec le graphe de visibilité des types sont significatifs. En effet, on observe que 85 % des classes d'objets identifiées, avec l'utilisation du graphe de visibilité des types, ont été acceptées. D'abord, l'expert nous a mentionné qu'il a développé le système `SBC1` en utilisant principalement des types complexes de données. Ensuite, il faut considérer qu'il s'agit uniquement des classes d'objets que notre approche a identifiées.

Les résultats obtenus avec le graphe de visibilité des données sont très intéressants. Malgré que 59 % des objets identifiés doivent être modifiés, les commentaires de l'expert sont extrêmement satisfaisants en ce qui concerne l'utilisation du graphe de visibilité des données. Généralement, les modifications devant être réalisées aux objets sont mineures. En effet, voici une liste non exhaustive des modifications devant être apportées aux objets identifiés.

- Enlever un attribut de type simple considéré comme une variable de travail.
- Enlever un attribut de type simple déjà compris dans un attribut de type complexe.
- Enlever un attribut qui fut passé en paramètre par un appel récursif de la fonction.
- Fusionner deux objets lorsque l'algorithme a trop minimisé les regroupements.
- Déplacer une méthode vers un autre objet.

- Diviser un objet en deux.
- Effectuer une focalisation manuelle d'une méthode appartenant à deux objets.

L'analyse des résultats obtenus avec le systèmes `SBC1` a permis de conclure que notre façon de calculer le seuil peut être amélioré. De fait, le seuil est calculé à partir du ΔIC maximum $-\epsilon$. Cette manière de calculer le seuil permet de regrouper les variables des fonctions ayant un ΔIC maximum. De plus, la nouvelle variable crée par ce regroupement possède les fonctions ayant un ΔIC maximum. Le problème survient lorsque des fonctions ont par hasard un ΔIC identique et qu'il s'agit du ΔIC maximum. Par conséquent, l'algorithme regroupe des variables qui n'ont aucun lien entre elles.

SBC1

Graphe	**Objets rejetés**	**Objets à modifier**	**Objets acceptés**
Graphe de références	0 %	20 %	80 %
Graphe de références amélioré	0 %	20 %	80 %
Graphe de visibilité des types	15 %	0 %	85 %
Graphe de visibilité des données	9 %	59 %	32 %

TAB. 4.4: Validation des résultats de SBC1.

4.4.2 La validation de SBC2

Le tableau 4.5 présente la validation des résultats obtenus lors de l'identification des objets dans le système `SBC2`. Ce tableau permet de constater que l'utilisation du graphe de références permet d'obtenir les meilleurs résultats. Par ailleurs, l'utilisation du graphe de références amélioré a permis d'identifier moins d'objets qu'avec l'utilisation du graphe de références. Par contre, le nombre d'objets que l'expert a rejetés est moindre avec l'utilisation du graphe de références amélioré qu'avec le graphe de références. Néanmoins, la

différence entre les résultats obtenus pour ces deux graphes n'est pas réellement significative.

L'utilisation du graphe de visibilité des types a permis d'identifier moins de classes d'objets avec le système SBC2 qu'avec le système SBC1. Cette différence importante des résultats est probablement causée par un style de programmation différent. En effet, contrairement au système SBC2, le système SBC1 a été programmé à l'aide de types abstraits de données. Donc, il est normal de trouver plus facilement des classes d'objets dans le système SBC1 que dans le système SBC2.

Les résultats obtenus avec l'utilisation du graphe de visibilité des données sont moins significatifs pour le système SBC2 que pour le système SBC1. De fait, avec le système SBC1, seulement 9 % des objets furent rejetés, tandis qu'avec le système SBC2 50 % des objets furent rejetés. Cette importante variation des résultats entre les deux systèmes est causée par une différence au niveau du style de programmation. En effet, le système SBC2 comporte un grand nombre d'indicateurs et de variables de travail ne devant pas faire partie des objets. De plus, l'expert a mentionné que certains objets étaient formés d'attributs de types simples déjà compris dans un attribut de type complexe appartenant au même objet. Finalement, l'expert a signalé que certains objets devaient simplement être fusionnés.

SBC2

Graphe	**Objets rejetés**	**Objets à modifier**	**Objets acceptés**
Graphe de références	30 %	30 %	40 %
Graphe de références amélioré	27 %	46 %	27 %
Graphe de visibilité des types	31 %	63 %	6 %
Graphe de visibilité des données	50 %	38 %	12 %

TAB. 4.5: Validation des résultats de SBC2.

4.4.3 La validation de SBC3

Le tableau 4.6 présente la validation des résultats obtenus lors de l'identification des objets dans le système SBC3. Les résultats obtenus avec le système SBC3 semblent meilleurs que ceux obtenus précédemment. En effet, il est possible d'observer que le pourcentage d'objets à modifier et d'objets acceptés est très significatifs par rapport aux systèmes précédents. L'expert a mentionné que plusieurs objets à modifier doivent simplement être fusionnés. En effet, à plusieurs reprise l'expert a suggéré de regrouper de deux à cinq objets. De plus, il est intéressant d'observer que c'est la première fois qu'un système obtient des résultats meilleurs avec le graphe de visibilité des données qu'avec le graphe de références. Ces résultats sont possiblement causés par une plus grande utilisation de variables globales déclarées localement. Il s'agit de variables locales à chaque fonction et passées d'une fonction à l'autre par l'utilisation de paramètres. En outre, l'expert a mentionné que le système SBC3 comporte une librairie de fonctions. Par conséquent, ce système comporte moins de variables déclarées globales.

SBC3

Graphe	**Objets rejetés**	**Objets à modifier**	**Objets acceptés**
Graphe de références	24 %	41 %	35 %
Graphe de références amélioré	24 %	47 %	29 %
Graphe de visibilité des types	14 %	44 %	42 %
Graphe de visibilité des données	3 %	51 %	46 %

TAB. 4.6: Validation des résultats de SBC3.

CONCLUSION

L'identification des objets dans un code source procédural est un processus complexe. En effet, ce processus doit être adapté selon le profil de l'application. Notre approche comprend quatre types de graphes pouvant être utilisés comme entrée à l'algorithme d'identification des objets. Il s'agit du graphe de références, du graphe de références amélioré, du graphe de visibilité des types et du graphe de visibilité des données. Les expérimentations permettent de conclure que l'utilisation du graphe de références est recommandable lorsqu'une application possède un nombre significatif de variables globales par rapport à son nombre de fonctions. Parfois, les objets obtenus sont de mauvaise qualité, car certaines fonctions ont été trop focalisées. Par conséquent, il est suggéré d'utiliser le graphe de références amélioré pour minimiser la focalisation des fonctions touchant à deux sous-graphes. L'utilisation du graphe de visibilité des types peut permettre d'identifier les classes d'objets. Par contre, les expérimentations démontrent que si le nombre de types complexes n'est pas significatif par rapport au nombre de fonctions de l'application, les classes d'objets identifiées possèdent un trop grand nombre de méthodes et doivent être modifiées. Le dernier type de graphe pouvant être considéré comme entrée à l'algorithme d'identification des objets est le graphe de visibilité des données. Ce graphe permet d'identifier les objets adéquatement si le nombre de variables de travail de type simple et le nombre d'indicateurs ne sont pas significatifs par rapport au nombre de fonctions. En effet, les expérimentations permettent de conclure que les objets identifiés avec le graphe de visibilité des données comportent habituellement trop d'attributs de type simple. Lorsque le choix du graphe est effectué, il faut utiliser l'algorithme d'identifica-

tion des objets pour décomposer le graphe en sous-graphes connexes. La décomposition du graphe en sous-graphes connexes permet d'obtenir des candidats objets. Chacun de ceux-ci est formé du nom de l'objet, d'un ensemble d'attributs et d'un ensemble de méthodes. Les attributs permettent de conserver l'état de l'objet, tandis que les méthodes permettent de modifier le comportement de l'objet.

Les forces de la solution

Notre approche nous permet de tenir compte du profil de l'application pour choisir le type de graphe qui sera utilisé en entrée à l'algorithme d'identification des objets. Il s'agit d'une amélioration face à l'approche présentée par [4] qui utilise seulement le graphe de références en entrée à l'algorithme d'identification des objets. Ensuite, l'approche que nous présentons est basée sur des heuristiques inspirées de la théorie des objets. En effet, l'approche est implantée en considérant les deux heuristiques suivantes : la considération du mode d'utilisation d'une variable globale par une fonction lors de la focalisation, et la détermination du seuil en utilisant le ΔIC maximum $-\epsilon$, permettant de minimiser la taille des objets. Il s'agit d'une autre amélioration face à l'approche présentée par [4] qui considère uniquement des heuristiques de la théorie des graphes. Enfin, notre approche est indépendante du domaine de l'application, car elle ne demande pas de modèles d'analyse de l'application pour fonctionner. Cela nous permet d'avoir une approche d'identification des objets automatisable.

Les faiblesses de la solution

Bien que notre approche soit automatisable, elle requiert un travail supplémentaire non négligeable pour obtenir un modèle de classes complet, c'est-à-dire un modèle représentant la hiérarchie des classes. De plus, bien que le profil de l'application puisse suggérer d'utiliser le graphe de références, il peut arriver que l'application ne possède pas

suffisamment de variables globales pour obtenir des résultats représentatifs. Par ailleurs, il est possible que le graphe de visibilité des données ne soit pas envisageable si le nombre de variables temporaires dans l'application est trop élevé. Par conséquent, il faut affiner le graphe de visibilité des données de manière à éliminer les variables de travail de type simple et les indicateurs.

Le prolongement des travaux

Nous présentons ci-dessous une liste non exhaustive des tâches devant être réalisées pour parvenir à effectuer la migration d'un code source procédural vers un code source orienté objets [20].

Filtrage ascendant des méthodes : optimiser le modèle orienté objets de l'application pour que les méthodes appartiennent à la bonne classe. Par exemple, si une classe `C2` hérite d'une classe `C1`, alors une méthode `M1` qui est propre à la classe `C2` peut se trouver dans la classe `C1`. La notion d'héritage accorde le droit à la classe `C2` d'utiliser la méthode `M1`. Par contre, comme la méthode `M1` est utilisée uniquement dans la classe `C2`, elle doit se retrouver obligatoirement dans celle-ci.

Conversion du code source : réorganiser le code source orienté objets de l'application. En effet, les retours de valeurs, les paramètres, les déclarations et les appels de chaque méthode doivent être implantés adéquatement.

Implantation des caractéristiques de l'orienté objets : implanter les constructeurs, les destructeurs, le polymorphisme et l'héritage.

Affectation des niveaux de protection : implanter les classes avec les étiquettes *public*, *protected* et *private*.

Regroupement dans des fichiers : diviser le code source de l'application en plusieurs fichiers. Habituellement, chaque classe doit avoir au moins deux fichiers : un pour la déclaration de la classe et un autre pour les méthodes de cette classe.

ANNEXES

Annexe A - Exemple1.cc

```
/*****************************************************************/
/*                                                               */
/*  AUTEUR:              Francois Dumont                         */
/*                                                               */
/*  DATE DE CREATION:    1 Avril 1997                            */
/*                                                               */
/*  NOM DU PROGRAMME:    Exemple1.cc                             */
/*                                                               */
/*  DESCRIPTION:         Ce programme C est utilise comme exemple */
/*                       dans le memoire de l'auteur.            */
/*                                                               */
/*****************************************************************/


/*==================================*/
/*                                  */
/*        STRUCTURES GLOBALES       */
/*                                  */
/*==================================*/
```

```
typedef struct
    {
      int  Nombre1;
      char Lettre1;
    } Structure1_Globale;

typedef struct
    {
      int  Nombre2;
      char Lettre2;
    } Structure2_Globale;

typedef struct
    {
      int  Nombre3;
      char Lettre3;
    } Structure3_Globale;

typedef struct
    {
      int  Nombre4;
      char Lettre4;
    } Structure4_Globale;

/*==================================*/
/*                                  */
/*        VARIABLES GLOBALES        */
/*                                  */
```

```
/*==================================*/
Structure1_Globale   Variable1_Globale;
Structure2_Globale   Variable2_Globale;


/*==================================*/
/*                                  */
/*            FONCTIONS             */
/*                                  */
/*==================================*/
      void    Fonction_1 ( void );
      void    Fonction_2 ( void );
      void    Fonction_3 ( Structure3_Globale, Structure4_Globale );
      void    Fonction_4 ( void );
      void    Fonction_5 ( void );

/*-----------------------------------------------------------------*/
/*                                                                 */
/*  DESCRIPTION:  Fonction principale                              */
/*                                                                 */
/*-----------------------------------------------------------------*/
int main()
    {
     Structure3_Globale Variable1_Locale;
     Structure4_Globale Variable2_Locale;

     Variable1_Globale.Nombre1 = 1;
     Variable1_Globale.Lettre1 = 'A';
```

```
    Variable2_Globale.Nombre2 = 2;
    Variable2_Globale.Lettre2 = 'B';

       Variable1_Locale.Nombre3  = 3;
       Variable1_Locale.Lettre3  = 'C';

       Variable2_Locale.Nombre4  = 4;
       Variable2_Locale.Lettre4  = 'D';

       Fonction_1();
       Fonction_2();
       Fonction_3(Variable1_Locale, Variable2_Locale);
}


/*-------------------------------------------------------------*/
void Fonction_1( void )
    {
       Variable1_Globale.Nombre1 = 10;
    }

/*-------------------------------------------------------------*/
void Fonction_2( void )
    {
       Variable1_Globale.Nombre1 = 20;
       Fonction_4();
       Fonction_5();
```

```
    }

/*-------------------------------------------------------------------*/
void  Fonction_3(Structure3_Globale Var1_F3, Structure4_Globale Var2_F3)
    {
        Variable1_Globale.Nombre1 = 30;

        Var1_F3.Nombre3 = 30;
        Var2_F3.Nombre4 = 40;
    }

/*-------------------------------------------------------------------*/
void Fonction_4( void )
    {
        Variable1_Globale.Nombre1 = 40;
    }

/*-------------------------------------------------------------------*/
void Fonction_5( void )
    {
        void Fonction_6 ( void );

        Variable1_Globale.Nombre1 = 50;

        Fonction_6();
    }

/*-------------------------------------------------------------------*/
```

```
void Fonction_6( void )
    {
        Structure1_Globale Variable3_Locale;

        Variable3_Locale.Nombre1 = 100;
    }
```

Annexe B - Exemple2.cc

```
/*****************************************************************/
/*                                                               */
/*  AUTEUR:              Francois Dumont                         */
/*                                                               */
/*  DATE DE CREATION:    13 Aout 1997                            */
/*                                                               */
/*  NOM DU PROGRAMME:    Exemple2.cc                             */
/*                                                               */
/*****************************************************************/


/*==================================*/
/*                                  */
/*        STRUCTURE GLOBALE         */
/*                                  */
/*==================================*/
typedef struct
    {
      int Nombre1;
      char Lettre1;
    } Structure1_Globale;

typedef struct
    {
      int Nombre2;
      char Lettre2;
    } Structure2_Globale;
```

```
typedef struct
    {
      int Nombre3;
      char Lettre3;
    } Structure3_Globale;

typedef struct
    {
      int Nombre4;
      char Lettre4;
    } Structure4_Globale;

/*==================================*/
/*                                  */
/*        VARIABLES GLOBALES        */
/*                                  */
/*==================================*/


/*==================================*/
/*                                  */
/*             FONCTIONS            */
/*                                  */
/*==================================*/
      void Fonction_1 ( Structure1_Globale, Structure2_Globale );
      void Fonction_2 ( Structure1_Globale, Structure2_Globale );
      void Fonction_3 ( Structure1_Globale, Structure2_Globale );
```

```
      void Fonction_4 ( Structure3_Globale, Structure4_Globale );
      void Fonction_5 ( Structure3_Globale, Structure4_Globale );

/*-------------------------------------------------------------*/
int main()
    {
      Structure1_Globale Variable1_Locale;
      Structure2_Globale Variable2_Locale;
      Structure3_Globale Variable3_Locale;
      Structure4_Globale Variable4_Locale;

      Variable1_Locale.Nombre1 = 1;
      Variable1_Locale.Lettre1 = 'A';

      Variable2_Locale.Nombre2 = 2;
      Variable2_Locale.Lettre2 = 'B';

      Variable3_Locale.Nombre3  = 3;
      Variable3_Locale.Lettre3  = 'C';

      Variable4_Locale.Nombre4  = 4;
      Variable4_Locale.Lettre4  = 'D';

      Fonction_1(Variable1_Locale, Variable2_Locale);
      Fonction_4(Variable3_Locale, Variable4_Locale);
}
```

```
/*-------------------------------------------------------------*/
void Fonction_1( Structure1_Globale Var1_F1, Structure2_Globale Var2_F1 )
    {
      Var1_F1.Nombre1 = Var1_F1.Nombre1 + 1;
      Var2_F1.Nombre2 = Var2_F1.Nombre2 + 2;

      Fonction_2(Var1_F1, Var2_F1);
    }

/*-------------------------------------------------------------*/
void Fonction_2( Structure1_Globale Var1_F2, Structure2_Globale Var2_F2 )
    {
      Var1_F2.Nombre1 = Var1_F2.Nombre1 + 1;
      Var2_F2.Nombre2 = Var2_F2.Nombre2 + 2;

      Fonction_3(Var1_F2, Var2_F2);
    }

/*-------------------------------------------------------------*/
void  Fonction_3( Structure1_Globale Var1_F3,
                  Structure2_Globale Var2_F3 )
    {
      Var1_F3.Nombre1 = Var1_F3.Nombre1 + 1;
      Var2_F3.Nombre2 = Var2_F3.Nombre2 + 2;
    }

/*-------------------------------------------------------------*/
void Fonction_4( Structure3_Globale Var1_F4, Structure4_Globale Var2_F4 )
```

```
    {
      Var1_F4.Nombre3 = Var1_F4.Nombre3 + 1;
      Var2_F4.Nombre4 = Var2_F4.Nombre4 + 2;

      Fonction_5(Var1_F4, Var2_F4);
    }

/*-------------------------------------------------------------*/
void Fonction_5( Structure3_Globale Var1_F5, Structure4_Globale Var2_F5 )
    {
      Var1_F5.Nombre3 = Var1_F5.Nombre3 + 1;
      Var2_F5.Nombre4 = Var2_F5.Nombre4 + 2;
    }
```

Annexe C - Exemple3.cc

```
/*****************************************************************/
/*                                                               */
/*  AUTEUR:              Francois Dumont                         */
/*                                                               */
/*  DATE DE CREATION:    2 Novembre 1997                         */
/*                                                               */
/*  NOM DU PROGRAMME:    Exemple3.cc                             */
/*                                                               */
/*****************************************************************/


/*==================================*/
/*                                  */
/*       DEFINITIONS GLOBALES       */
/*                                  */
/*==================================*/
#define MAXDIM 99
typedef int ELEM_T;
typedef int BOOL;
struct list_struct  {
        ELEM_T node_content;
        struct list_struct * next_node;
};


/*==================================*/
/*                                  */
```

```
/*        VARIABLES GLOBALES       */
/*                                 */
/*=================================*/
ELEM_T          stack_struct[MAXDIM];
ELEM_T          queue_struct[MAXDIM];

list_struct     list;

int             stack_point;
int             queue_head;
int             queue_tail;
int             queue_num_elem;


/*-----------------------------------------------------------------*/
int main()
{
        /* La fonction "main" de ce programme ne contient aucune
           instruction, car ce programme a simplement comme objectif
           de montrer les liens entre une "stack", une "queue" et
           une "list". */
}

/*-----------------------------------------------------------------*/
void stack_push(int element)
{
        stack_point     = 1;
        stack_struct[0] = element;
```

```
}

/*-------------------------------------------------------------*/
ELEM_T stack_pop()
{
        stack_point     = 2;
        stack_struct[0] = 2;
        return(2);
}

/*-------------------------------------------------------------*/
ELEM_T stack_top()
{
        stack_point     = 3;
        stack_struct[0] = 3;
        return(3);
}

/*-------------------------------------------------------------*/
BOOL stack_Empty()
{
        stack_point     = 4;
        return(1);
}

/*-------------------------------------------------------------*/
BOOL stack_full()
{
```

```
        stack_point     = 5;
        return(1);
}

/*-----------------------------------------------------------------*/
void queue_insert(int element)
{
        queue_struct[0] = element;
        queue_head      = 6;
        queue_num_elem  = 6;
}

/*-----------------------------------------------------------------*/
ELEM_T queue_extract()
{
        queue_struct[0] = 7;
        queue_tail      = 7;
        queue_num_elem  = 7;
        return(7);
}

/*-----------------------------------------------------------------*/
BOOL queue_Empty()
{
        queue_num_elem  = 8;
        return(1);
}
```

```
/*-------------------------------------------------------------*/
BOOL queue_full()
{
        queue_num_elem  = 9;
        return(1);
}

/*-------------------------------------------------------------*/
void list_add(int element)
{
        list.node_content = element;
}

/*-------------------------------------------------------------*/
void list_elim(int element)
{
        list.node_content = element;
}

/*-------------------------------------------------------------*/
BOOL list_is_in()
{
        list.node_content = 10;
        return(1);
}

/*-------------------------------------------------------------*/
BOOL list_empty()
```

```
{
        list.node_content = 11;
        return(1);
}

/*-------------------------------------------------------------*/
void global_init()
{
        stack_point       = 12;
        queue_head        = 12;
        queue_tail        = 12;
        queue_num_elem    = 12;
        list.node_content = 12;
}

/*-------------------------------------------------------------*/
void stack_to_list()
{
        stack_point       = 13;
        stack_struct[0]   = 13;
        list.node_content = 13;
}

/*-------------------------------------------------------------*/
void stack_to_queue()
{
        stack_point     = 14;
        stack_struct[0] = 14;
```

```
        queue_struct[0] = 14;
        queue_head      = 14;
        queue_num_elem  = 14;
}

/*-------------------------------------------------------------*/
void queue_to_stack()
{
        queue_struct[0] = 15;
        queue_tail      = 15;
        queue_num_elem  = 15;
        stack_point     = 15;
        stack_struct[0] = 15;
}

/*-------------------------------------------------------------*/
void queue_to_list()
{
        queue_struct[0]   = 16;
        queue_tail        = 16;
        queue_num_elem    = 16;
        list.node_content = 16;
}

/*-------------------------------------------------------------*/
void list_to_stack()
{
        list.node_content = 17;
```

```
        stack_point        = 17;
        stack_struct[0]    = 17;
}

/*-------------------------------------------------------------*/
void list_to_queue()
{
        list.node_content = 18;
        queue_struct[0]    = 18;
        queue_head         = 18;
        queue_num_elem     = 18;
}
```

BIBLIOGRAPHIE

[1] B. J. ARSENEAU and T. SPRACKLEN. An artificial neural networks based software reengineering tool for extracting objects. *IEEE*, pages 3888–3893, 1994.

[2] B. J. ARSENEAU and T. SPRACKLEN. Reengineering software modularity using artificial neural networks. In *Worl Congress on Neural Networks-San Diego*, 1994.

[3] G. CANFORA, A. CIMITILE, and M. MUNRO. RE^2 : Reverse-engineering and reuse re-engineering. *Software Maintenance : Research and Practice*, 6:53–72, 1994.

[4] G. CANFORA, A. CIMITILE, and M. MUNRO. An improved algorithm for identifying objects in code. *Software - Practice and Experience*, 26(1):25–48, Janvier 1996.

[5] E. CHIKOFSKY and J. H. CROSS. Reverse engineering. In John-Wiley, editor, *Encyclopedia of Software Engineering*, pages 1077–1084. J.J. Marciniak, 1994.

[6] P. COAD and E. YOURDON. *Object-Oriented Analysis.* Yourdon Press computing series, second edition, 1990.

[7] H. GALL and R. KLÖSCH. Program transformation to enhance the reuse potential of procedural software. In *ACM Symposium on Applied Computing*, pages 99–104, Phoenix, USA, Mars 1994.

[8] H. GALL and R. KLÖSCH. Finding objects in procedural programs : An alternative approach. In IEEE Computer Society Press, editor, *2nd International Working Conference on Reverse Engineering (WCRE'95)*, pages 208–216, Toronto, Canada, Juillet 1995.

[9] H. GALL, R. KLÖSCH, and R. MITTERMEIR. Architectural transformation of legacy systems. In William Griswold, editor, *Workshop on Program Transformation for Software Evolution*, Avril 1995.

[10] H. GALL, R. KLÖSCH, and R. MITTERMEIR. Object-oriented re-architecturing. In *Fifth European Software Engineering Conference (ESEC'95)*, pages 499–519, Septembre 1995.

[11] R. E. JOHNSON and B. FOOTE. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, Juin/Juillet 1988.

[12] M. KOUBARAKIS, J. MYLOPOULOS, M. STANLEY, and A. BORGIDA. Telos : Features and formalization. Technical Report Technical Report KRR-TR-89-4, Dept. of Computer Science, University of Toronto, 1989.

[13] C. L. MCCLURE. *The Three Rs of Software Automation.* Prentice-Hall, inc., 1992.

[14] E. MENIF. Des techniques de slicing pour l'analyse statique de programmes. Technical Report Avril-97, Dept. informatique de l'Université Laval et le CRIM, 1997.

[15] E. MERLO, I. MC ADAM, and R. DE MORI. Source code informal information analysis using connectionnist models. *Neural Networks*, pages 1339–1344, 1993.

[16] S. PAUL and A. PRAKASH. Supporting queries on source code : A formal framework. Cet article n'a pas été publié.

[17] R. S. PRESSMAN. *Software Engineering a Practitioner's Approach.* McGraw-Hill, 1992.

[18] S. RUGABER. Program comprehension for reverse engineering. In *AAAI Workshop on AI Automated Program Understanding.* Georgia Institute of Technology, Juillet 1992.

[19] R. W. SCHWANKE and S. J. HANSON. Using neural networks to modularize software. *Machine Learning*, 15:137–168, 1994.

[20] J. SHIN. Migration of structured procedural C programs into object-oriented C++ based on code reuse. Master's thesis, University of Pennsylvania, Décembre 1996.

[21] I. SOMMERVILLE. *Software Engineering.* Addison-Wesley, 1995.

[22] T. WASHBURNE, R. STACHOWITZ, J. HAWLEY, and H. ROMSDAHL. Automatic classification of software modules with probabilistic neural networks. In *IEEE International Conference on Neural Networks*, pages 3894–3899, 1994.

[23] M. WEISER. Program slicing. *IEEE Trans. Software Engineering*, SE-10(4):352–357, Juillet 1984.

[24] G. WHITTINGTON, T. SPRACKLEN, and J. MACRAE. Applications of artificial neural networks to reverse software engineering. *Intelligent Engineering Systems through Artificial Neural Networks*, pages 163–169, 1994.

[25] A. S. YEH, D. R. HARRIS, and H. B. REUBENSTEIN. Recovering abstract data types and object instances from a conventional procedural language. In IEEE Computer Society Press, editor, *2nd International Working Conference on Reverse Engineering (WCRE'95)*, pages 227–236, Toronto, Canada, Juillet 1995.

[26] K. ZÉROUAL. *Université de Sherbrooke, notes de cours IFT722 - Génie Logiciel*, Hiver 1996.